Introduction to Cyber Technologies

A course developed by the Cyber Technologies Academy

Sandia National Laboratories, Livermore, CA

This text should be accompanied by an exercise disk image on which to perform the specified exercises.

For more information about the Cyber Technologies Academy (CTA) or to download the disk image, visit our website at https://share.sandia.gov/cta.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85.

AUTHORS

Jeremy Erickson
jericks@sandia.gov

Craig Shannon
cdshann@sandia.gov

Kina Winoto
kwinoto@sandia.gov

Steve Hurd
sahurd@sandia.gov

CW Perr
cwperr@sandia.gov

Levi Lloyd
lloyd@sandia.gov

Document version: 1.01.00

NOTICE:

For five (5) years from 12/18/2014, the United States Government is granted for itself and others acting on its behalf a paid-up, nonexclusive, irrevocable worldwide license in this data to reproduce, prepare derivative works, and perform publicly and display publicly, by or on behalf of the Government. There is provision for the possible extension of the term of this license. Subsequent to that period or any extension granted, the United States Government is granted for itself and others acting on its behalf a paid-up, nonexclusive, irrevocable worldwide license in this data to reproduce, prepare derivative works, distribute copies to the public, perform publicly and display publicly, and to permit others to do so. The specific term of the license can be identified by inquiry made to Sandia Corporation or DOE.

NEITHER THE UNITED STATES GOVERNMENT, NOR THE UNITED STATES DEPARTMENT OF ENERGY, NOR SANDIA CORPORATION, NOR ANY OF THEIR EMPLOYEES, MAKES ANY WARRANTY, EXPRESS OR IMPLIED, OR ASSUMES ANY LEGAL RESPONSIBILITY FOR THE ACCURACY, COMPLETENESS, OR USEFULNESS OF ANY INFORMATION, APPARATUS, PRODUCT, OR PROCESS DISCLOSED, OR REPRESENTS THAT ITS USE WOULD NOT INFRINGE PRIVATELY OWNED RIGHTS.

Any licensee of This software has the obligation and responsibility to abide by the applicable export control laws, regulations, and general prohibitions relating to the export of technical data. Failure to obtain an export control license or other authority from the Government may result in criminal liability under U.S. laws.

Table of Contents

Lesson 1

In lesson 1, we are going to learn the basics of the Linux terminal.

Overview of Linux

Why do we use Linux? What can a Linux machine do that a Windows computer can't? Who uses Linux? Where did Linux come from?

Assignment number one: Go write a report on the origins and history of Linux, the pros and cons of Linux vs. other operating systems, and … you've already stopped listening, haven't you? Yep. We didn't feel like writing this report any more than you would want to read it, so we'll keep it short.

There is one name you should know, however: **Linus Torvalds**. We will simply point you to his Wikipedia page. Please click the link and read approximately 1-3 paragraphs.

http://en.wikipedia.org/wiki/Linus_Torvalds

Why are we learning Linux? Because although the majority of desktop machines may run Windows, Linux makes up most of the backbone of the Internet. The majority of the servers hosting the websites you use every day (Google, Facebook, Twitter, etc.) are all run on top of the Linux operating system, and there are a lot of things you can do with Linux that are difficult or impossible with Windows.

Introduction to the Linux command-line

When people talk about using Linux, they usually think about using the Linux command-line interface. In this class, we're going to spend almost all of our time using the terminal.

Hopefully you've got our exercise environment set up in front of you. Log in with username `student` and password `student`.

Navigating your way around the terminal

To open a terminal, either:

- Click the Activities menu at the top of the screen. Scroll down to the "Terminal" icon and click it.
- Press Ctrl+Alt+T (Note: this does not work on all versions of Linux, but does on our systems)

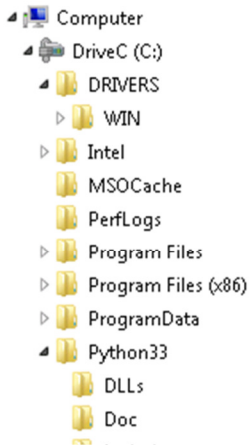


Figure 1.1: What the folder structure looks like in Windows

You should see a **prompt** that says `student@cta-student:~$`

Let's break down what this means:

- `student` is your **username**.
- The `@cta-student` refers to your **hostname**, cta-student.
- The colon (`:`) separates your hostname from your **current directory**.
- In this case, it's a tilde (~), which is a symbol that represents your **home directory** (`/home/student/`).
- Finally, the `$` signifies that you are **not root** (root is the admin user). If you were root, the last symbol would be a `#` (pound) symbol. This will be important later.

Directory structures

The command-line is useful for many things, but the simplest use case is file and directory management. Put simply, you can move files around, create directories, delete stuff, etc.

Directories in Linux are similar to folders in Windows (shown in Figure 1.1). They are ordered in a tree structure. To get to *higher* directories, you go *up* or *out*. You can also go *down* or *into* a lower-level directory.

In Linux, the top-level directory is `/` (slash). Your home directory is `/home/student/`, which looks like this:

- /
    - home/
        - student/

Let's try:

- `ls`

The "`ls`" command **lists** the contents of your current directory (mnemonic: **l**i**s**t).

Recall: What directory are you in right now?

You should see some words appear after running the `ls` command. Those are the files and directories within your current directory. On this system, directories are blue, and files are other colors (probably white).

One of the directories you should see is `lesson1`. Let's go into that directory:

- `cd lesson1`

The `cd` command **changes your current directory** (mnemonic: **c**hange **d**irectory). The `lesson1` argument specified to which directory you wanted to change. Note that, unless you specify an *absolute path* (e.g. one starting with `/`, such as `/home/student/lesson1`), it will attempt to find the directory from your current directory. This is called a *relative path*.

Now try to go back. You may think that since you were previously in a directory named `student` that you could use `cd student` to go back. If you try this, you will get:

```
bash: cd: student: No such file or directory
```

This is telling you that it can't find the `student` subdirectory under directory `lesson1`. To go back *up* a directory, you need to use the special name `..` ("dot dot").

- `cd ..`

Great, now that we're back in your home directory, let's go back to `lesson1` again.

- `cd lesson1`

Now try `ls` again.

- `ls`

You should see a directory called `exercise1`.

- `cd exercise1`
- `ls`

You should see a directory called `house`.

- `cd house`
- `ls`

You should see a new set of directories and files named after various pieces of furniture.

Exercise: Find the money stashed somewhere in the house

Bonus: How much money?

Head back to `/home/student/lesson1/` when you're done.

Manipulating files and directories

Let's learn a couple more commands.

- `ls`

You may notice that there is an `exercise1` directory, and an `exercise3` directory, but no `exercise2`. Let's create it:

- `mkdir exercise2`

The `mkdir` command **makes a new directory** (mnemonic: **m**a**k**e **dir**ectory).

- `ls`

You should now see the new `exercise2` directory. Notice that it made the new directory *in* your current directory.

Exercise: Using `cd`, `ls`, and `mkdir`, create the following directory structure:

- `exercise2/`
  - `projects/`
    - `cta/`
    - `school/`
  - `documents/`
    - `taxes/`

    - homework/
- notes/
    - school/

Great. Now let's learn some more commands.

- `cd /home/student/lesson1/exercise3/`
- `ls`

You should see a directory `poor` and a directory `money`.

- `mv poor rich`
- `ls`

You just **moved** the `poor` directory to the `rich` directory (mnemonic: **m**o**v**e). Since there wasn't a `rich` directory before, this just renamed the `poor` directory to `rich`. You can use the `mv` command to rename directories and files. The syntax is:

```
mv <source> <destination>
```

- `mv money rich`
- `ls`

You just **moved** the `money` directory to the `rich` directory. Since there was already a `rich` directory, you moved the `money` directory *inside* the `rich` directory. In this case, the `mv` command did not rename the `money` directory

- `cd rich`
- `ls`

You should be able to see that the `money` directory resides here now.

Moving files and directories is similar to Windows's cut-and-paste.

WARNING – Be careful when you move files, since moving a file on top of another file will overwrite it. When you move files into directories, make sure there isn't already a file of the same name in the directory or else it will be overwritten.

Go back up to the main `exercise3` directory.

- `cd ..`
- `ls`

Now, you should also see a file named `hokey`. Let's use the **copy** command to make a new file called `pokey` (mnemonic: **c**o**p**y).

```
cp <source> <destination>
```

- `cp hokey pokey`

The `cp` command will behave the same as the `mv` command, but rather than renaming it, will actually create a brand new copy of the file. Take a look.

- `ls`

You should see both `hokey` and `pokey` in the `exercise3` directory now.

When copying or moving a large file, the `cp` command will often take a long time, whereas the `mv` command will complete very quickly. Why is this?

Finally, let's learn how to delete unwanted files and directories.

- `rm hokey`
- `ls`

You should see that the `hokey` file has now been removed. The `rm` command will **remove** (delete), files (mnemonic: **re**move).

WARNING – Be very careful when removing files and directories. By default, there is no confirmation dialogue. Type the wrong thing and you could end up deleting things you didn't intend to.

Let's remove the `empty` directory:

- `rm empty`

You should see:

```
rm: cannot remove 'empty': Is a directory
```

Hm. So it looks like we cannot use the `rm` command to remove directories. (You actually can with the `-r` argument (recursive), but that's potentially more dangerous, so we won't do that yet).

There is another command we can use called `rmdir` that **removes a directory** (mnemonic: **re**move **dir**ectory).

- `rmdir empty`
- `ls`

You should now see that the `empty` directory is gone.

Let's try to use `rmdir` to remove the `full` directory:

- `rmdir full`

```
rmdir: failed to remove 'full': Directory not empty
```

Ah hah! `rmdir` is saving us from making a mistake. `rmdir` can only be used to remove empty directories so we don't accidentally delete the files inside. In this case, we really want to remove the `full` directory, so let's see what's inside and delete it first.

- `ls full`

Notice that we can give `ls` the name of the directory of which we want to list the contents. In this case, it should have displayed the contents of the `full` directory, and you should see there is a single file called `yum`.

- `rm full/yum`
- `ls`
- `ls full`

This time, we can delete the `yum` file without having to change directories to `full` first. Notice that the `yum` file gets deleted, but the `full` directory is still present. Let's wrap this up and delete the `full` directory now.

- `rmdir full`

Exercise: Using `cp`, `mv`, `rm`, and `rmdir`, make the `exercise3/` directory look like this:

Note: entries ending with a slash (/) are directories. Entries not ending with a slash are files.

- exercise3/
  - rich/
    - money/
  - pokey
  - new/
    - car
    - shoes/
      - laces
      - tongue
      - sole
    - pets/
      - lion/
        - mane
        - paws
      - unicorn

Hint: Since you don't know how to create new files yet, you should copy `pokey` to make the files.

Hint2: We're not showing you how to use it, but the `tree` command may come in handy for visualizing your directory structure.

Editing files

Learning how to manipulate the directory structure is all well and good, but hopefully it's leading somewhere, right?

Next, we'll learn about how to create and edit files. To start, let's use a simple text editor that's very similar to Notepad.

- `cd`

Note: the `cd` command with no other options will change your current directory to your home directory, `/home/student`.

- `cd lesson1/exercise4/`

Note: you don't really need to type the slash at the end of the directory name. We're just including it for clarity.

- `ls`

Notice the file named `fillmeupwith.txt`? Let's do just that.

- `gedit fillmeupwith.txt`

A window that looks similar to Notepad should have just popped up on your screen. Go ahead and enter in the phrase "text text text" into the `gedit` window (mnemonic: **G**NOME **Edit**or), then press the Save button and close `gedit`.

Now, `fillmeupwith.txt` should contain the text, "text text text". Let's double-check that it does.

- `cat fillmeupwith.txt`

The `cat` command allows you to **concatenate** files together, but in its simplest use, it simply prints the contents of a file to the console. Concatenate means *to link together*. You should see "text text text" appear on the next line.

Let's see what copying the file does when it has content.

- `cp fillmeupwith.txt somemore.txt`
- `cat somemore.txt`

You should see that `somemore.txt` now contains the same "text text text" as `fillmeupwith.txt` does.

Exercise: Explore the `exercise4/backwards/` directory and make the contents of each file equal to the name of the file spelled backwards.

For instance, a file named "rennid" should have the content "dinner" when you're done.

Running programs from the command-line

Huh? Haven't you been running programs this whole time?

Well, yes. But so far, the commands you've learned have all been built-in to Linux. Pretty soon, you'll be writing your own programs and wanting to execute them from the command-line. For instance, check out the contents of directory `exercise5/`.

- `cd`
- `cd lesson1/exercise5/`
- `ls`

You should see a file named `script.sh`. This is a **shell script**. A shell script is a kind of program that is made out of plain text. Take a look, but be sure not to change the file, because we will run it soon.

- `gedit script.sh`

Close the `gedit` window when you're done.

Let's run this program the same way we've run the other programs we've learned so far (`cd`, `ls`, `mkdir`, etc.):

- `script.sh`

```
bash: script.sh: command not found
```

What? How come it can't find the command. IT'S RIGHT THERE!

- `ls`

SEE?! IT'S RIGHT THERE!

So, what's happening is that bash can't find the `script.sh` script because even though it's in our current directory, by default bash doesn't look for commands in the current directory. To get it to run our program, we need to explicitly tell it where the command is located:

- `/home/student/lesson1/exercise5/script.sh`

This should run, printing out a line saying "Congratulations on running your first script!"

We could also run it like this:

- `./script.sh`

A single dot, "`.`" refers to our **current directory**, just as a double dot, "`..`" refers to our **parent directory**, up one level. Consequently, typing `./script.sh` tells bash to look in the current directory (`/home/student/lesson1/exercise5/`) for a file to run named `script.sh`.

Exercise: Run the `checkexercises.sh` script. It will check your work on the previous exercises. Go back and fix any issues it reports before moving on.

Lesson 2

In Lesson 2, we are going to learn some more advanced features of the Linux terminal, and why it is so powerful.

The manual

If you ever forget how a command works, or want to learn about its features and options, you can read its manual with

- `man <command>`

For example:

- `man ls`

This will bring up a manual page of the ls command. To exit the manual, press q.

Everything-is-a-file concept

In Linux there is an idea that "everything is a file". But what is "everything", and why does "everything" have to be a file? Everything in this case includes traditional files such as documents – but also hard drives, keyboards, and network interfaces. Almost all of the input/output resources in Linux are presented to us as a stream of bytes provided by the filesystem. What this gives us is the ability to use the same set of utilities across all of these resources. While this may seem counterintuitive, it will become more apparent as you familiarize yourself with the internals of Linux.

Searching for files

Over time your home directory will undoubtedly become filled with more and more files, and you will eventually forget where exactly you placed the file you're looking for. Linux provides the `find` command to help locate files. Find can be a very complex utility, so we will only cover some simple examples.

- `find /home/student/ -name lost.file`

This command searches for the file `lost.file` starting in the directory `/home/student/`.

- `find ~ -name "*.txt"`

This command searches for all files with the extension `.txt` starting in our home directory (`~`).

Exercise: Find things

Use the `find` command to find the following things:

1. A file with the size of 400 bytes in the `lesson2/exercise1/` folder
2. All of the hidden directories in the `lesson2/exercise1/` folder

- Hint: in this exercise, the hidden directories are named `.hidden`
- Except for one hidden directory, can you find it?

Hint: remember to use the man pages.

Hidden Files

In Linux, files starting with a period are called hidden files. By default, `ls` will not display them. To see them, you must use the `-a` option.

```
ls -a
```

Text manipulation

Now that we are able to create, edit, move, and delete files, we need to understand how to work with the text actually contained in these files. It is common on Linux machines to regularly deal with very large log files and complex configuration files, so it is important to understand how to efficiently parse what you need. The word *parse* means *to divide and identify parts based on their relationship to one another*.

Table 2.1 contains some of the most common and helpful text manipulation utilities. Remember that it is often necessary to combine more than one of these utilities to achieve your desired output, and to do that you need to utilize pipes (covered in the next section).

Table 2.1: Text manipulation commands

`grep`	searches for a particular word or string (sequence of characters)
`head`	prints the first N lines of a file or stdin
`tail`	prints the last N lines of a file or stdin
`cut`	remove sections from each line of files
`sort`	sort lines of text files
`wc`	prints the number of newlines, words, and bytes in files
`uniq`	report/omit repeated lines

While manipulating text is great, we need to use a different program if we want to actually *search* through files. `grep` is considered the Swiss army knife of text searching, and is one of the most frequently used tools by many Linux administrators. `grep` allows you to search text files for specified patterns.

```
grep "test" <file>
```

The above command will search the file `<file>` for the string `test` and print any occurrences it finds.

Linux pipes

Before we can fully utilize the text manipulation utilities, we first need to discuss **pipes** in Linux. Pipes give two processes (programs) the ability to communicate. Pipes consist of a read end and a write end. Normally, when a program writes to the console, it writes to a special stream called **stdout** (standard

output). When a program wants input, it can read from a special stream called **stdin** (standard input). A pipe connects one process's stdout to another process's stdin.

Pipes can be created in the command line with the | character. A simple example is given below.

```
cat test.txt | grep "hello"
```

The above example takes the output of the `cat` command and sends it to the input of the `grep` program.

Redirect to file

You can redirect output from stdout directly into a file like this:

```
find ~ -name "*.txt" > textfiles.txt
```

Be careful, this will overwrite the file you write into. If you use two alligator brackets (>>), you will *append* to the file instead:

```
echo "some text" > newfile.txt
echo "some more text" >> newfile.txt
```

You can also redirect a file directly into stdin:

```
grep "hello" < test.txt
```

Exercise: Play with some data

Use the `lesson2/exercise2/salaries.txt` file to try and solve the following questions. Save your commands so you can explain them later! Use the `man` pages to learn how to use the text manipulation commands.

Lets do one together:

1. How many unique employee salaries are there and what are they?
    - First, lets just print out all the employee salaries:
        - `cat salaries.txt`
    - Next, lets sort the list by dollar amount:
        - `cat salaries.txt | sort -k 4`
        - The `-k` starts the sort on the fourth field
    - Finally, lets get rid of duplicate salaries:
        - `cat salaries.txt | sort -k 4 | uniq -f 3`
        - The `-f` skips the first three fields when looking for matching lines
    - Now we know what all the unique salaries are, but lets count them:
        - `cat salaries.txt | sort -k 4 | uniq -f 3 | wc -l`

Now you try:

1. Print out ONLY the interns' information. Don't print out any manager, programmer, or technician information.

2. Print out ONLY the line with the highest salary.
3. Who is the 5th salary listed and print out ONLY that person's information. Don't print any other line besides the 5th one.
4. Sort the employees based on last name and print this sorted list.
5. Print out ONLY employee names, sorted by first name. Don't print out salary or job title information.

For the next part of this exercise, we will need `exercise2/salaries_additional.txt`. This file has some of the same information as `salaries.txt`, but also contains additional employee information. Your last task is to combine the files `exercise2/salaries.txt` and `exercise2/salaries_additional.txt` together with duplicate information removed to create a master combined list of salaries.

Wildcards and special characters

There are a number of special characters that the Bash shell (the Linux terminal) recognizes. For starters, the *space* is considered a special character. Imagine you want to create a new file called

```
space invaders.txt
```

Let's create it using the `touch` command

- `touch space invaders.txt`
- `ls`

Look at what happened. We created two new files, one called `space` and the other called `invaders.txt`. We don't want that. Let's delete those files.

- `rm space`
- `rm invaders.txt`

To create the single file `space invaders.txt` we need to **escape** the special space character to use a literal space character instead.

- `touch space\ invaders.txt`
- `ls`

The backslash (`\`) is called the escape character, and can be used to ignore the effects of other special characters. For instance, if you wanted a literal backslash in the name of a file, you could do something like this.

- `touch back\\slash.txt`
- `ls`

The first backslash escapes the special effect of the second backslash.

There is another special character that gets used very often, and that's the asterisk (`*`). It is called a **wildcard** because it will become whatever character or multiple characters necessary to match as many filenames as it can. For instance:

- `cd /home/student/lesson2/exercise3/`
- `ls *.txt`

The `*` will expand to match the filename of any file ending in `.txt`, so this `ls` command will list all the files in the current directory ending in `.txt`.

Exercise: Find the secret message in `exercise3/`

There are a lot of files in `exercise3/`. To find the secret message, remove all files that satisfy one or more of the following properties:

1. end in `.txt`
2. contain `zzz` in their name
3. start with the letter `i`

Then use `cat` to display the contents of all the remaining files at once.

You'll know you've completed the exercise when it makes sense:
http://www.youtube.com/watch?v=dQw4w9WgXcQ

SSH and Bandit exercises

Congratulations! You now know your way around the Linux command-line. One last command we should cover is `ssh`. `ssh` stands for **secure shell** and is the usual way we run commands on remote machines.

In a typical scenario, you may be working on your computer for a while, and then at some point need to perform some action on another computer. Let's call this remote computer `chaz`. Assuming that `chaz` is hooked up to the network, you can log in remotely using the command:

```
ssh student@chaz
```

In this example, `student` is the **username** you're logging in as and `chaz` is the **host**. Most of the time, the host will be a domain name like `chaz.mywebsite.com` or even an IP address like `192.168.1.7`.

If this had been a real system, a "`Password:`" prompt would pop up for you to type in your password. Note that when you type in your password, you won't see any characters, dots, or asterisks being typed. This is a security feature so that your neighbor can't see how many characters long your password is.

After logging in, your terminal (or, shell) prompt should change to show that you are now logged in to a new machine. You'll be reset to your home directory on the remote machine, and commands you enter will run on the remote machine.

When you want to drop the `ssh` connection and return to your own machine, type `exit`.

Exercise: SSH into Bandit Labs

Open a web browser and navigate to http://overthewire.org. Over The Wire provides a number of excellent "Wargames" that give you practice with computer science and cybersecurity topics. The one of interest to us right now is Bandit. Click the Bandit link on the left side of the page.

You will see a series of Levels on the left-hand side of the page. Each level will introduce new Linux commands to you and expect you to use them to solve a puzzle. Upon completion of the puzzle, you will be given the password to the next Bandit Level. By completing all of the challenges, you can reach Level 25 and win. If you aren't sure how to proceed, make sure to read the man page of any commands you aren't familiar with, as well as the Helpful Reading Material each level provides.

To get to Level 0, our first objective is to log in to the bandit server using `ssh`. Remembering how to use `ssh` from above, and since they give us a username of **bandit0**, a host of **bandit.labs.overthewire.org**, and an initial password of **bandit0**, the command we will use to access Level 0 is:

- `ssh bandit0@bandit.labs.overthewire.org`

While not part of this class, we recommend you play a few of the Bandit levels for fun!

Lesson 3

In Lesson 3, we are going to learn about users and access control – who has permission to do what to which files and directories.

Access control

Access control is, quite simply, the ability to control access to something. You can control who has access to files, directories, and devices, based on who may need to know certain information or interact with certain things.

For each file, there is a matrix relationship between three types of entities and three types of permissions:

Entities

User – The user that owns the file
Group – The group of users that have group access to the file
Other – Everyone else

Permissions

Read – Read a file or look inside a directory
Write – Modify a file or modify a directory's contents
Execute – Run a file or enter a directory

Each file and directory is owned by one user and one group. That user has access to the User permissions and users in that group have access to the Group permissions. Everyone has Other permissions.

Using `ls` with the `-l` option (that's an L, not a 1), you can list the contents of your current directory and print out lots of extra information, including file and directory permissions.

- `cd /home/student/lesson3/exercise1/`
- `ls -l`

```
student@cta-student:~/lesson3/exercise1$ ls -l
total 32
drwxr-xr-x 2 student student 4096 Sep  5 10:14 1
d--x------ 2 student student 4096 Sep  5 10:14 2
dr-------- 2 student student 4096 Sep  5 10:14 3
---x------ 1 student student   44 Sep  5 10:14 a
-r-------- 1 student student   22 Sep  5 10:14 b
--w------- 1 student student   22 Sep  5 10:14 c
-rw-r--r-- 1 student gamers  1284 Sep  5 10:14 gamesecrets.txt
-rw------- 1 student student 3123 Sep  5 10:14 passwords.txt
student@cta-student:~/lesson3/exercise1$
```

Notice that each file and directory has some combination of `drwxrwxrwx` permissions. Dashes specify that the permission is *not* present.

`d`: Whether the file is a directory or not
First `rwx`: User permissions
Second `rwx`: Group permissions
Third `rwx`: Other permissions

In looking at the first line:

```
drwxr-xr-x 2 student student 4096 Sep 5 10:14 1
```

We see that `1` is a directory owned by the `student` user and the `student` group, with read/write/execute permissions for the User (student), read/execute permissions for the Group (student), and read/execute permissions for everyone else. It is expected that directories will typically have the execute permission set so that users of the system can enter the directory.

What do `--w-------` permissions mean?

If you want to **change the permissions** of a file or directory, you can use the `chmod` command.

- `cd 1/`

Remove the User's read permission:

- `chmod u-r file.txt`

Add the write permission for the Group:

- `chmod g+w file.txt`

Add the execute permission for Other:

- `chmod o+x file.txt`

Give everyone(User, Group, and Other) the execute permission:

- `chmod +x file.txt`

Exercise: Explore the exercise1 directory. What files can you read from? What files can you write to? What files can you execute? What directories can you explore? Modify the permissions until you find everything.

Users and Groups

On a Windows family machine, there may be multiple users with different accounts. On a Linux machine, there are many different user accounts, both for people and various system services that should only have access to certain files.

Each user will have a home directory in `/home/`. By default, other users will typically be able to read most files in other users' home directories, but not write to them. It is important to make sure that your file permissions don't give other users access to information you want to keep secret.

To see the **users that are currently logged on to the system**:

- `users`

To see the **groups you are in**:

- `groups`

At this point, you can't **add or delete users or groups** (you need root access, or to be part of the sudo group), but the commands to do so are:

- `adduser edward`
- `deluser edward`
- `addgroup spiky`
- `delgroup spiky`

To add/remove a user to/from a group:

- `adduser edward spiky`
- `deluser edward spiky`

To change the owner and group of a file:

- `chown edward file.txt`
- `chgrp spiky file.txt`

Or

- `chown edward:spiky file.txt`

Got it? Great. (No really, use this section as a reference once you get root access)

Root, Sudo, and Switching Users

There is a *special group* called `sudo` that allows use of the `sudo` command. `sudo` makes the following command **run as the root user**. The root user has permission to do anything, including changing owners, groups, and permissions on files owned by another user.

- `sudo adduser frank`

The `sudo` command will require you to enter in your password again.

If you want to **switch users**, you can use the `su` command like so:

```
su frank
```

The system will prompt you for Frank's password, and after entering it, you will be logged in as Frank. Once you are Frank, you can change to Frank's home directory with `cd`. Frank's home directory is `/home/frank/`, just as yours is `/home/student/`. Go ahead and investigate the other users' home directories as the `student` user. There should be lots of files you can access, and lots more you can't. Try changing the permissions on a file you don't own.

What if you don't have someone's password, but you have sudo privileges?

```
sudo su frank
```

Root can switch to any other user without needing their password, so you can use `sudo` to gain access.

Exercise: Add the student user to the sudo group

This is a fairly long exercise that involves finding a way to access the accounts of several other users on your system (Alice, Bob, Charlie, Diane, and Mallory). Your starting point can be found in `/home/student/lesson3/exercise2/`.

Once you get `sudo` access, log out and log back in again so that it takes effect.

Installing applications

We've all had to install an application on our computers. Without the ability to install new or update old applications, our computers would quickly become out-of-date. In Windows, most applications are installed using an installer program that you would double-click. In Linux, applications and other system components are usually installed via a **package manager**.

There are two main ways to install an application on Linux machines. We can use a **package manager**, which is a lot like an installer program in that it takes care of the heavy lifting, or we can build things **from source code**. Building from source code requires configuring, compiling, and moving files where needed. Package managers abstract this away from us and deal with all of that behind the scenes.

First, we'll discuss different varieties of package managers. Depending on the Linux distribution, the package manager may have a different name. Several are listed in Table 3.1.

Table 3.1: Package managers

Distribution	Package Manager Name	Command
Debian / Ubuntu	Advanced Packaging Tool	`apt-get`
Red Hat / Fedora	Yum	`yum`
Arch	PacMan	`pacman`
OpenSuse	Zypper	`zypper`

Before we get started on the commands to install an application via a package manager, what is a package? You've likely seen a package before and just didn't know it had a name. On a Mac, you may

have seen the `.pkg` extension, which is an example of a package. Packages are types of archives (archives are files that have been compressed, such as `.zip` files) that contain files and metadata. In Debian/Ubuntu, packages used to install programs have `.deb` extension, Red Hat/Fedora uses `.rpm`, and Arch uses `.aur`.

So why do we use package managers? We use them to resolve dependencies, meaning install all needed pre-requisites to our desired software. They also keep track of what has been installed and where, what is no longer in use, and what needs to be updated. Without them, we would have to build everything from source code. Package managers pull applications from a **repository**, a datastore of applications and dependencies. A package manager can pull from more than one repository and in the case of apt-get, it maintains a list of repositories at `/etc/apt/sources.list`.

For now, we will focus on one package manager, `apt-get`. A few useful and common commands are shown in Table 3.2.

Table 3.2: apt-get commands

Update the list of known packages	`apt-get update`
Install a package	`apt-get install <package>`
Upgrade installed packages that have new versions	`apt-get upgrade`
Remove a package	`apt-get remove <package>`
Remove a package and purge all related configuration files	`apt-get remove --purge <package>`
Remove all packages that were previously installed as dependencies but are no longer needed	`apt-get autoremove`
Related to `apt-get`: Search for packages	`apt-cache search <search term>`

Exercise: install a new package

Note: installing packages requires root access

- `sudo apt-get update`
- `apt-cache search tux`
- `apt-cache search tux | grep "super"`
- `sudo apt-get install supertux`
- `supertux`

Go ahead and play supertux for a minute or two. Learning is fun, right?

- `sudo apt-get remove --purge supertux supertux-data`

Notice that you can install or remove multiple packages at once by separating them with spaces. Package managers aren't just for games though. Let's install Wireshark, a network traffic analyzer we'll be using in the next lesson.

- `sudo apt-get install wireshark`

During the installation, when it prompts you about whether non-superusers should be able to capture packets, say YES. Then add the `student` user to the `wireshark` group.

- `sudo adduser student wireshark`

Log out and log back in so the new group takes effect.

Processes

Every time you launch an application, a **process** is launched. A process is an instance of a program being run on a machine. Processes take up a number of resources managed by the operating system, including memory, descriptors (e.g. network and file connectors), and security attributes. Processes can launch other processes, which is called **spawning** a child. Each process has a unique process ID number, called a **PID**. We view and interact with PIDs instead of the name of the application because applications can be run multiple times simultaneously, which could lead to naming conflicts.

There are a few common command line tools that we use to view and stop active processes. They are shown in Table 3.3.

Table 3.3: Process management commands

`ps`	Print snapshot of running processes
`top`	Print detailed information about the top running processes
`kill <pid>`	Stop the running process with pid <pid>

Drilling down, a few common commands within `ps` are shown in Table 3.4.

Table 3.4: ps command options

`ps –e`	Shows every process
`ps –f`	Show the full listing of processes
`ps –u <user>`	Shows all the processes running under user <user>
`ps –U <user>`	Shows all the processes except those running under user <user>

So what do we actually use these commands for? First of all, `ps` works really well with pipes and `grep`. For instance, if we wanted to see if an instance of supertux is running, we could do:

```
ps –e | grep “supertux”
```

```
student@cta-student:~$ ps -e | grep "supertux"
 4868 pts/1    00:00:18 supertux
student@cta-student:~$
```

In addition, let’s say we have a rogue instance of supertux that we are unable to quit using the normal method. We would use the above command to get the PID associated with supertux (the first number).

In this example, the PID of supertux is 4868. So we use

```
kill 4868
```

to kill the rogue instance of supertux.

Note that each instance of supertux is assigned a different PID so if you followed along, you'll likely have a different PID.

Exercise: Kill the rogue processes

Navigate to `/home/student/lesson3/exercise3/` and follow the instructions.

Hint: The **`diff`** command may come in very handy (use "`man diff`" to figure it out).

Be careful. If you kill processes the system needs to run, your computer could become unstable or even crash entirely.

Lesson 4

In this lesson, we will cover in introduction to the network stack and the Data Link layer of the OSI model.

The OSI Model and an introduction to how networks work

The OSI Model is the customary networking model used to describe how machines successfully communicate with each other. **With no exercises in this first section, I know it's tempting to look at the pictures and skim, but this is a really important section.** We're going to go back over each layer of the OSI model in depth over the next few classes, but it's important you get an idea of how all the layers work together to pass messages from machine to machine.

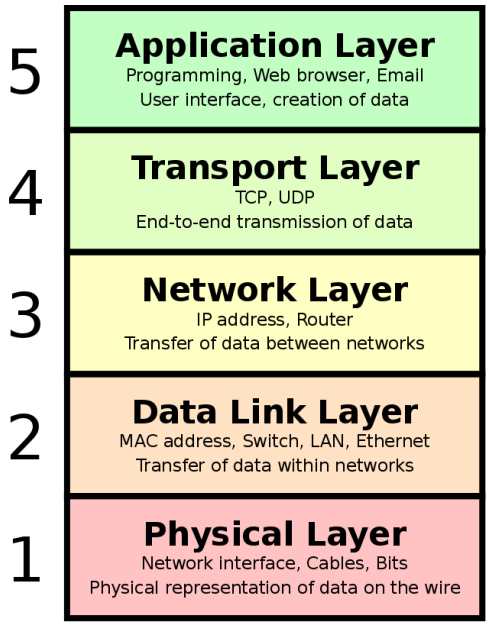


Figure 4.1: The OSI model

As shown in Figure 4.1, the OSI Model is broken up in to five major layers, each with its own roles and responsibilities. This model is not a perfect representation of all the different aspects of networking, but helps logically categorize different sets of tools and protocols together into a common theme. For instance, Layer 2, the Data Link Layer, generally encompasses all the constructs, tools, and protocols that deal with the transfer of data between machines that are all on the same local network. When we want to talk about moving data *between* networks, we go up to Layer 3, the Network Layer.

Application (Layer 5)

The purpose of the Application Layer is to allow applications to send messages to each other. Your web browser, twitter app, and even the software keeping your car running are all applications that send messages. The application layer encompases the messages your application creates and the description of where you want to send them and passes them down to the Transport Layer for delivery. By abstracting away the actual transport of the message to a different layer, you can write applications easily without having to deal with the underlying complexity of the full networking stack.

Say I write a chat program. I write it in my favorite programming language, Python (shameless plug – check out our Programming courses!). The code I write to interact with the network might look something like this (don't worry if you don't know Python, just read the comments in bold text):

```
sock = socket.socket(...) # create a new network socket
sock.connect((“google.com”,23)) # connect to google.com on port 23
sock.send(bytes(“Here’s some data” * 1000, ”utf-8”)) # send data
response = sock.recv(4096) # receive 4096 bytes of data
sock.close() # close the connection to google.com
```

In this program, I create a text string (underlined above) of "Here's some dataHere's some dataHere's some data…" that goes on for 16,000 characters. This message takes up 32,000 bytes (16 characters repeated 1000 times, times 2 bytes per character), so it's a pretty substantial message. This message will be handed from my application to the operating system's network stack for processing.

Transport (Layer 4)

Generally, networks limit the maximum message size you can send in a single packet to around 1500 bytes. Since my message is much larger than this (32,000 bytes), it's going to need to be broken up into chunks, or packets, of data. Each packet will be passed to the Network Layer to be sent over the wire individually and reassembled on the other side into the original message. The Transport layer is responsible for breaking up the message and reassembling it at the other end, (usually) making sure not to lose any packets.

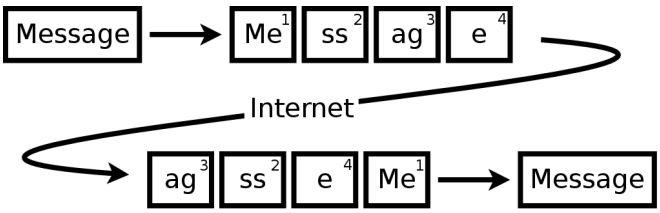


Figure 4.2: Message reconstruction

Network (Layer 3)

Once the message has been broken up into packets, it is up to the Network layer to send each packet to its destination. Each packet can now be treated independently.

At the Network layer, we address packets to another entity via **Internet Protocol (IP) address**, similar to how we address paper mail to entities via street address. Through the magic of DNS (which is actually an Application Layer protocol, and explained in a later chapter), the machine will discover that google.com corresponds to an IP address of 74.125.28.139 and will address each packet to 74.125.28.139.

Why can't we simply address each packet to google.com?

Although we know the eventual destination of the packet, it's exceptionally unlikely that our machine is on the same local network as the machine with that IP address. Since our machine doesn't know the route to the destination address, it will send the packet to a router (192.168.1.1) that may have a better idea about where to send it next. That router will pass the packet on to the next router, which in turn will send it on to the next router, and so on until the packet reaches a router that knows the destination machine (74.125.28.139). Eventually, the destination machine will receive the packet. When all the packets have arrived, the machine at google.com will reassemble them (layer 4) and deliver them to the application (layer 5).

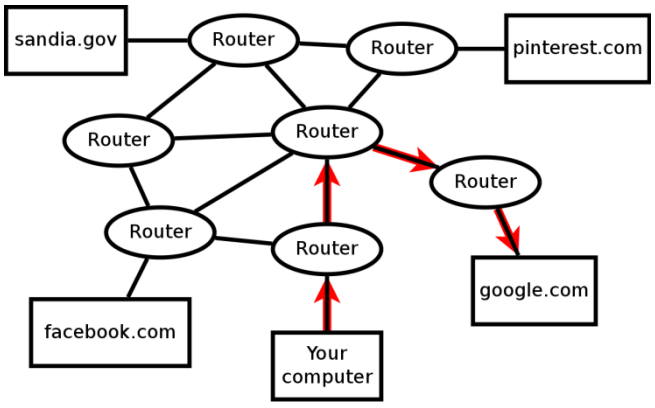


Figure 4.3: Routing

Data Link (Layer 2)

While the Network layer is responsible for routing packets from router to router to get them to their eventual destination, it leaves the process of sending the packet from machine to machine to the Data Link Layer. Layer 3 is responsible for communication between networks, while Layer 2 is responsible for communication *within* a network. As we continue to progress down the layers, notice that we move from a high level view to the specifics of how messages are sent from device to device.

In the very first hop of the packet's journey, your machine (machine B in Figure 4.4) wants to send the packet to your router so that your router can forward it on to its eventual destination. At Layer 2, we no longer deal with IP addresses. Instead, we use **Machine Address Code (MAC) addresses** to identify machines. Each network interface (e.g. Ethernet port) will have a unique MAC address. When a machine sends a packet on to the wire, the machine's MAC address is encoded in the packet as the *source address*. A switch connects multiple devices together, watches the source addresses contained in packets it sees, and builds up a table associating each MAC address with a particular switch port.

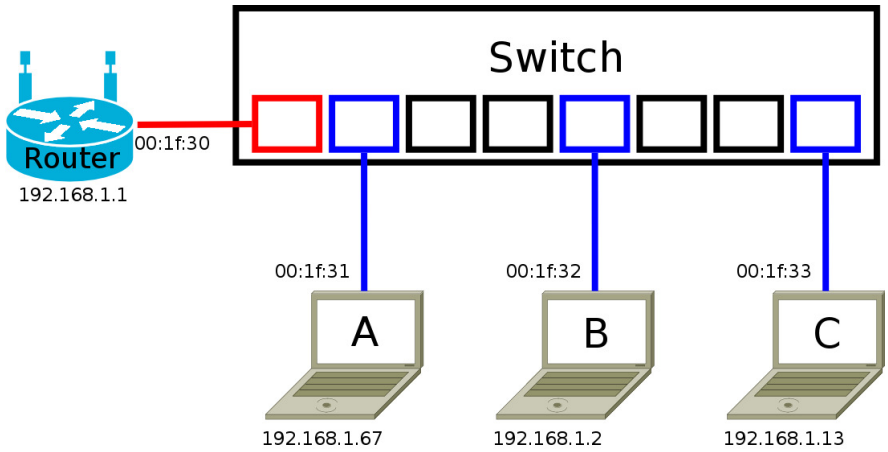


Figure 4.4: Example home network

To send a packet to the router, your machine would also encode a *destination address* in the packet it sends that corresponds to the router's MAC address. In this example, you would send a packet with a source address of `00:1f:32` and a destination address of `00:1f:30`. Since your network interface is directly plugged in to the switch, your packet would be delivered to the switch. The switch then uses the destination address to decide which port to forward your packet on. If it doesn't know, it will forward your packet out all ports.

Physical (Layer 1)

After all the complicated logical steps of the above layers, the physical layer is conceptually much simpler. The Physical Layer's purpose is simply to transfer a series of bits as quickly as possible between two "connected" devices. In an Ethernet network, this means modulating the voltage on the wire between two network cards to represent 0's and 1's. A wireless connection will behave similarly, although obviously without needing the wire. Each data link packet, or frame, will be encoded into a

series of bits and then transferred across the physical medium. When it reaches the device on the other side, it will be decoded and either processed for its hop to the next device or passed up to higher layers.

Although the Physical Layer is a huge subject area, we will pretty much ignore it in this class. The Physical Layer is grounded in electrical hardware whereas this class focuses on software.

Traversing the layers

So, what does it look like when a message passes through all the layers and gets sent out on the wire? See Figure 4.5.

At each layer, some data is added to the growing packet. This additional data is called a **header**. Each layer knows how to parse the data in its own header, but ignores the higher layers' headers, treating them as data. Figure 4.5 shows only some of the most pertinent features of each layer's header. In reality, each header may have 10 or more fields that describe different properties of the packet. You'll learn more about these fields in a college-level networking course, but they aren't particularly useful for the understanding we're shooting for in this class.

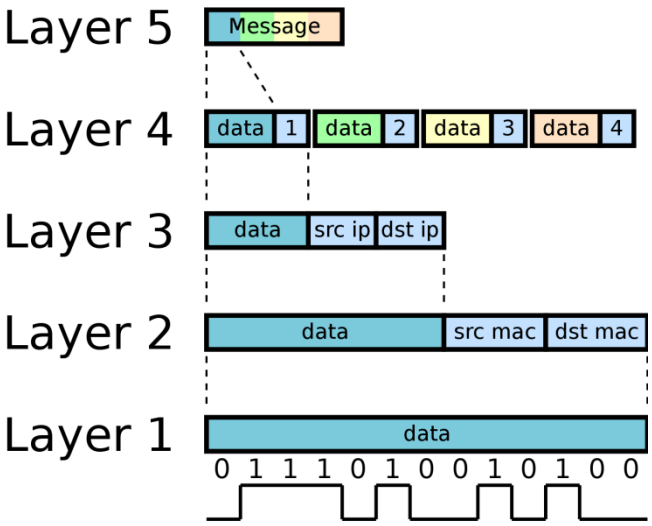


Figure 4.5: Packet encapsulation

Take a break

That was a lot to take in all at once. Feel free to take a breather before diving into the next section.

Don't worry, now that we have an overview of how networks work, we will spend the next few lessons slowly covering each of the network layers and their associated networking protocols.

Layer 2: MAC addresses and the Local Area Network (LAN)

Layer 2 Packet Structure: Frame

The Data Link layer allows us to transfer data between network nodes using messages encapsulated in a **frame**.

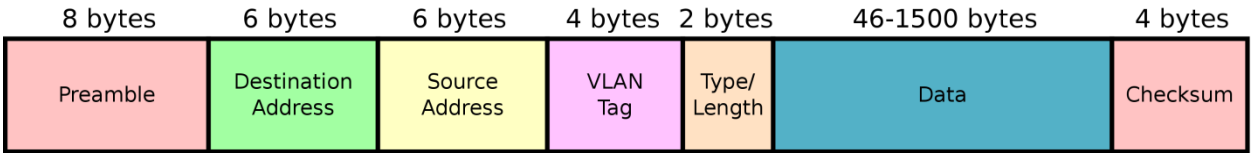


Figure 4.6: Frame structure

The important fields to notice from Figure 4.6 are:

- Destination Address
- Source Address
- Data

At layer 2, the source and destination addresses are MAC addresses. A MAC address is generally a unique hardware identifier for the network interface card. The source address refers to the sender's MAC address, whereas the destination address refers to the MAC address of the network interface you want to receive the packet. Note that other machines may also see the packet, but are supposed to drop it or forward it on instead of processing it if the destination MAC address doesn't match their MAC address.

The data field is where the message goes, but the message will also be surrounded by layer 3 and layer 4 encapsulation fields, such as the source and destination IP addresses, within the data field.

Exercise: Identify your MAC address and network card vendor

Run the command:

- `sudo ifconfig`

Note: you will not be able to find the `ifconfig` command if you do not use sudo.

You should see a series of network devices on the left, with information about each on the right. For instance, `eth0` is the first Ethernet device and `wlan0` is the first wireless device. Additional Ethernet or wireless devices would take the form `eth1`, `eth2`, or `wlan1`, `wlan2`, etc. If you don't see `eth0` or `wlan0`, try the `-a` option, to print out *all* devices, even if they aren't enabled.

- `sudo ifconfig -a`

```
student@cta-student:~$ sudo ifconfig
eth0      Link encap:Ethernet  HWaddr 2c:41:38:0d:32:57
          inet addr:192.168.50.111  Bcast:192.168.50.255  Mask:255.255.255.0
          inet6 addr: fe80::2e41:38ff:fe0d:3257/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:14 errors:0 dropped:0 overruns:0 frame:0
          TX packets:73 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3543 (3.4 KiB)  TX bytes:10305 (10.0 KiB)
          Interrupt:20 Memory:d4500000-d4520000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:28 errors:0 dropped:0 overruns:0 frame:0
          TX packets:28 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:2304 (2.2 KiB)  TX bytes:2304 (2.2 KiB)
```

Identify the MAC addresses of `eth0` and `wlan0` under the "Hwaddr" field. The first three bytes identify the manufacturer of the network card. For instance, in the above screenshot, the MAC address of `eth0` is `2c:41:38:0d:32:57` and the vendor can be identified with `2c:41:38`. You may also notice that one or more of your network devices have an IP address under the "inet addr" field.

Use Google or another website such as macvendorlookup.com to look up the vendor that made your network cards.

ARP: Discovering MAC addresses

Before machine A can send a packet to machine B, it has to know B's MAC address. How does it discover B's MAC address? It simply asks.

ARP stands for the Address Resolution Protocol, and it specifies a way for a machine to resolve a machine's MAC address from its IP address. At layers 3 and up, we operate on IP addresses, so at layer 2, we already have machine B's IP address available to work with.

In the following scenario, machine A has MAC address 11:11:11:11:11:11 and machine B has MAC address 22:22:22:22:22:22.

ARP works as follows:

1. Machine A wants to send a packet to IP address 192.168.1.1, but doesn't know which physical machine address (mac) is associated with that IP. In particular, there is no entry for 192.168.1.1 in machine A's ARP table.
2. Machine A sends a broadcast request to anyone on the same Local Area Network (LAN) asking, "Who is 192.168.1.1? Tell 11:11:11:11:11:11". This broadcast packet will have a special broadcast *destination* address of FF:FF:FF:FF:FF:FF, which all machines will listen to, and a *source* address of 11:11:11:11:11:11, since it is sent from machine A.
3. When the packet reaches machine B it will accept the packet because it is a broadcast packet. If machine B has IP address 192.168.1.1, it will reply by sending a directed packet to 11:11:11:11:11:11 (source address 22:22:22:22:22:22 and destination address

11:11:11:11:11:11) saying, “192.168.1.1 is at 22:22:22:22:22:22”. If machine B has a different IP address, it will simply ignore the ARP request.

4. Machine A receives the response packet, and since it is destined for 11:11:11:11:11:11, machine A accepts it. Machine A then updates its ARP table with an entry associating IP 192.168.1.1 with MAC address 22:22:22:22:22:22.
5. Finally, machine A will be able to send its original packet destined for 192.168.1.1 to 22:22:22:22:22:22.

Exercise: Virtual Networks and ARP traffic

Navigate to `/home/student/lesson4/exercise1/`. Inside, you will find a script named `start.sh`. Run the script as root.

- `./start.sh`

This script will not print anything out. Several things will happen in the background:

- A new network interface card named `veth0` will be created. This is a virtual Ethernet interface that we will use for this exercise. Check it out with `ifconfig`. You can interact with it the same way you would interact with `eth0` or `wlan0`. What IP address does this interface have? What MAC address?
- You will see a few other new network devices, such as `mega_bridge` and `mega_tap0`. For now, simply ignore these devices. They form the virtual network that this exercise runs on.
- A virtual machine (VM) has been created and connected to the virtual network. This VM is, for all intents and purposes, the same thing as a real computer connected to the network. The difference is that both the VM and `veth0` are virtual, meaning they run entirely in software within your computer. This is a handy technique we use in this course’s exercises to avoid asking you to purchase five or six additional computers and run around physically stringing wires between them.

In this exercise, the network topology looks like this:

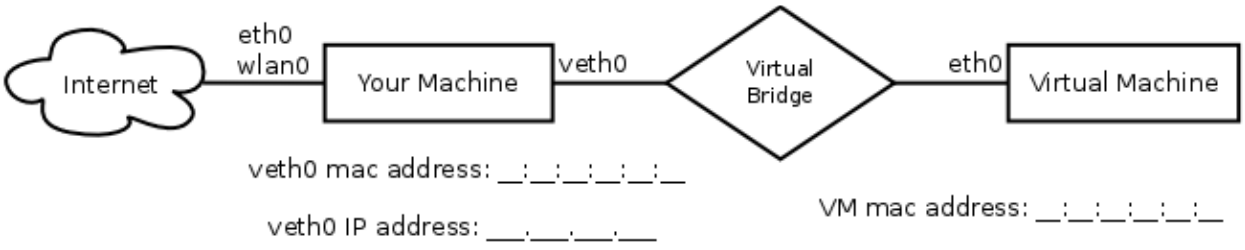


Figure 4.7: Exercise 1 topology

Fill the `veth0` IP and MAC addresses from the output of the `ifconfig` command in to the blanks in Figure 4.7.

Hm… But how do we find the MAC address of the VM? Let's take a look at the packets on the virtual network directly. Remember how last lesson we installed an application called wireshark? Go ahead and run it.

- `wireshark`

You should see a new window pop up. To listen to network traffic, you need to specify a network interface to listen on. In this case, we want to listen for packets from the virtual machine, so we need to listen to an interface connected to the virtual machine's network.

Which of the network interfaces on your machine do you want to listen to?

Click on the interface you want to listen to, and then click Start.

You will see the window change. Frequently, the top area that shows packets is only a few lines tall. You may need to grab the slider and drag it down to see the packets on the network. Note: the packets shown below are different from those you will see in this exercise.

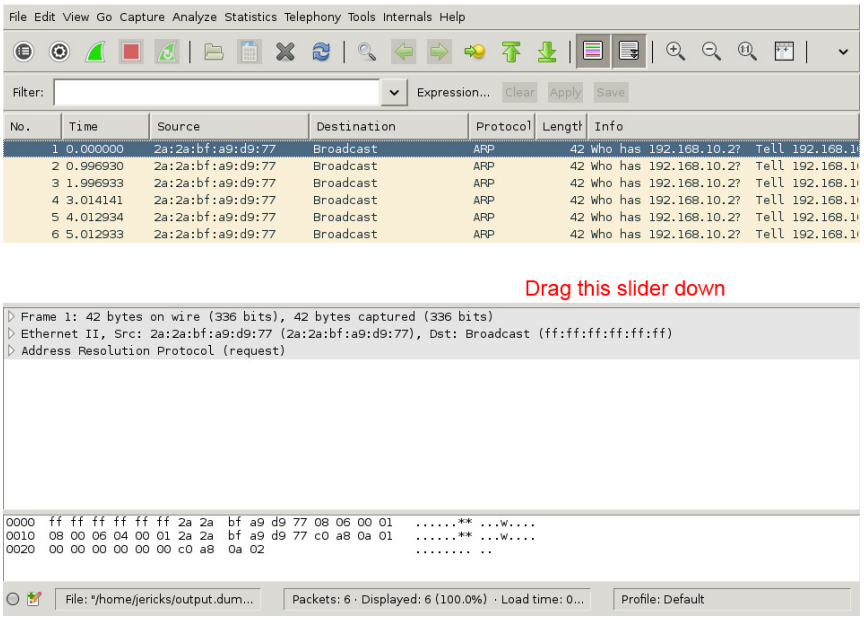


Figure 4.8: Wireshark - make sure to drag the slider down

Congratulations! You are capturing raw network traffic. You can use wireshark to dig in deeper and look at everything all the way down to the bytes being transferred over the wire.

Each line in the top area is a packet. If you look at the packets that should be appearing (if not, make sure you picked the right network interface), you can see that wireshark pulls out the source and destination addresses for each packet and displays them for you. If wireshark understands the network protocol of the packets it sees, it often also puts a helpful description of what the packet means in the "Info" column.

Now, based on what you're seeing, can you find the MAC address of the virtual machine?

What does the virtual machine appear to be doing every 5 seconds?

What is your computer doing in response?

Switches, bridges, and hubs. Oh my!

Looking at the network topology of the previous exercise, there was a funny diamond shape in the middle of the virtual network labeled "Virtual Bridge". What is a bridge? Well, the short answer is that a **bridge** is essentially the same thing as a switch implemented in software, or in other words, a virtual switch.

Note: In this document, we say "Virtual Bridge" for extra clarity, even though it's really just a bridge. It's a little like saying "ATM machine" (Automated Teller Machine machine). Sometimes it just makes things a little clearer.

Okay, so that didn't really answer what a bridge does or how it works, but at least we've knocked down our list of new terms from three things to two: Switches and Hubs. Both switches and hubs serve the same role in a network – to connect devices – but they behave very differently.

Figure 4.9: Network Switch

An example of a switch is shown in Figure 4.9. A hub would look almost identical ("Switch" would instead be spelled H-U-B).

Let's focus on hubs first. A **hub** allows you to connect two, three, four, or more devices together on the same network. You can connect hubs to each other to add more ports to plug computers in to. A hub is very simple. When it receives a packet on one interface, it forwards the packet out of each of its other interfaces.

In the topology shown below, let's assume the virtual bridge will act like a hub. In this topology, the hub has machines plugged into three of its ports.

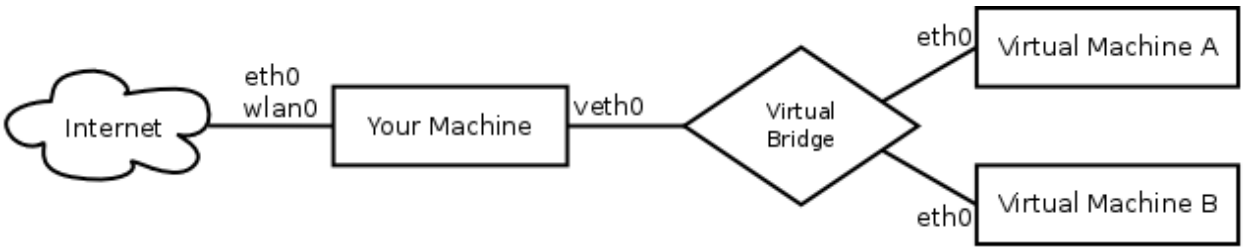


Figure 4.10: Exercise 2 network topology

VM A sends a packet to VM B. The hub will read the packet from VM A's port and then send the packet out on VM B's port, but it will also send the packet out the port that your `veth0` interface is connected to. It is up to your machine's network interface to recognize that the packet had a destination MAC address that doesn't match `veth0`'s MAC address and to drop the packet. How inefficient!

Well, what if we could get a hub to only send the traffic out of the ports it needs to? Congratulations, that's a **switch**.

The way a switch works is that every time it sees a packet, that packet has a source address field, and the switch keeps track of which source addresses appear in which ports. Say, the switch sees a packet arrive on port 2 with source address 00:11:00:11:00:11. The switch will keep a track of the fact that if it sees another packet with *destination* address of 00:11:00:11:00:11, it only needs to send it out of port 2 for the packet to reach its destination, instead of sending it out of all ports. Over time, a switch will build up a **MAC address table** that looks similar to the one shown in Table 4.1.

Table 4.1: Example switch MAC address table

Port	Address
2	`00:11:00:11:00:11`
3	`11:11:11:11:11:11`
0	`00:00:00:f4:f4:f4`
2	`e2:e2:e2:c4:c4:c4`
2	`00:11:22:00:11:22`

When the switch receives a packet, if the packet's destination address is in its MAC address table, it can send it out of that port and only that port.

What does the switch need to do if there is no entry in its MAC address table?

These days, hubs are not used in production environments. You can't even purchase them in stores anymore. In comparison, switches reduce network congestion and generally stop traffic from reaching unauthorized entities with no downsides. However, it's important to understand how hubs work because:

1. Switches are frequently still configured to act like hubs for learning, debugging, and monitoring purposes.
2. Wireless connections (WiFi) behave similarly to a hub network in several ways.

Exercise: Hub vs. Switch

Navigate to `/home/student/lesson4/exercise2/`. Inside, you will see two scripts, `hub.sh` and `switch.sh`. Each of these scripts works very similarly to `start.sh` from the previous exercise. They both create the virtual network shown in Figure 4.10, now with two virtual machines attached to the virtual network. However, in one of them, the virtual bridge has been configured to behave like a hub.

Run each of the scripts to set up each network, and after running each script, listen to the network traffic with wireshark.

In one of the network configurations, you should see periodic network traffic, but not in the other one. Why the difference?

What are the IP and MAC addresses of the two virtual machines?

What protocol does wireshark say the two machines are communicating over?

It turns out that the two machines are running a chat program and their users are having a conversation. What are the two users' names?

Lesson 5

In this lesson, we will cover the Network layer of the OSI model: IP addresses, subnetting, routing, and common layer 3 protocols.

The IP Address

In the Data Link Layer (layer 2), the address we used to differentiate between network devices was the MAC address. In the Network Layer (layer 3), we use a different kind of address you may be more familiar with, the IP **address**.

Whereas a MAC address is a built-in hardware address for a network device, an IP address is an identifier that is assigned to a network device. That is, IP addresses can, and often do, change. Where MAC addresses tend to refer to specific pieces of physical network hardware, IP addresses are better for referring to logical entities. They can also be assigned dynamically. Your laptop may get an IP address of 192.168.1.4 on your home network, but if you pick it up and carry it to a friend's house, it may get assigned an IP address of 192.168.1.37. Although it's somewhat rare, you may also occasionally see a network interface with multiple IP addresses assigned to it. That sort of flexibility requires a different kind of identifier than a MAC address.

Typically, unless you are diagnosing network problems at a very low level, you will use IP addresses to refer to other machines on your network, or on other networks.

Lets focus on a couple of tools that will be instrumental in the next exercise.

ifconfig

You should be pretty familiar with `ifconfig` by now to look at your network interfaces. Back in lesson 4, you should have noticed that in addition to a MAC address, most of your interfaces should have had an IP address assigned. `ifconfig` is an invaluable tool for discovering the current state of network connectivity on a computer. In the next exercise, you will use it to discover each computer's IP address(es).

ssh

Back in lesson 2, we used `ssh` to remotely log in to the bandit server. We're going to start using `ssh` extensively in exercises from this point on in the class, so to recap, the syntax we'll use is:

```
ssh [user@]host
```

For example:

```
ssh student@192.168.1.1
```

Or, if we are already logged in as a user named "student", we can save some typing with:

```
ssh 192.168.1.1
```

tcpdump

Last lesson, we used `wireshark` to look at packets on the network. `wireshark` is a great tool for looking at network traffic, but unfortunately cannot be used from within a terminal. Instead, we're going to show you a new tool called `tcpdump`. `tcpdump` does the same thing as `wireshark`, it lets you look at live network traffic, but it doesn't have an interactive interface.

Typically, you run `tcpdump` as root, and specify a network interface on which to begin capturing network traffic:

```
sudo tcpdump -i <interface>
```

`<interface>` in this case refers to one of your network interfaces, such as `eth0`, `wlan0`, or for our exercises, `veth0`. Remember, you can use `ifconfig` to find your network interfaces.

By default, `tcpdump` tries to resolve domain names, which can cause slowdowns. Personally, I prefer to turn off both domain name and port resolution and run `tcpdump` like this:

```
sudo tcpdump -nni <interface>
```

Exercise: Traverse the network

In this exercise, you will traverse several networks we have created for you. Navigate to `lesson5/exercise1` and run `start.sh`. The following virtual network will be created (Note:switches are omitted for brevity):

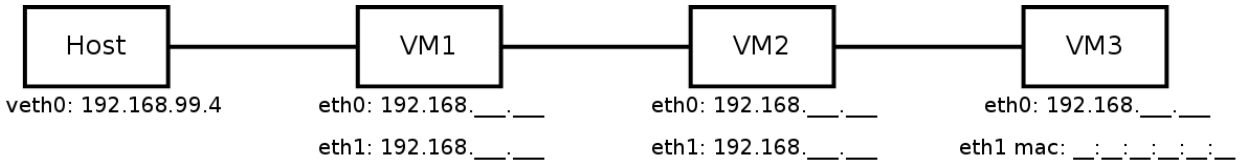


Figure 5.1: Exercise 1 network topology

The Host and VM1 share a network, as do VM1 and VM2, and VM2 and VM3. The Host cannot connect directly to VM2 or VM3.

Your goal is to use `ssh` to traverse the network and **find the MAC address of `eth1` on VM3**. Each network segment connecting two machines will be on a different subnet. Each VM has a user `student` with a password of `student`.

Hint: To find the IP address of the next VM, you will need to use `ifconfig` to see what network each interface is on and `tcpdump` to listen for ARP traffic on the network.

Networks

When dealing with IP addresses, it's important to understand the concept of a **network**. In this class, a network is a collection of network devices (switches, routers) and endpoints (computers) that can *directly communicate with each other*.

For instance, in the last exercise, there were three separate networks: (1) The network connected to the Host and VM1, (2) the network connected to VM1 and VM2, and (3) the network connected to VM2 and VM3. The Host could not communicate directly with VM2 or VM3 because it did not share a network with them.

IP address breakdown

Let's take a look at the construction of the IP address:

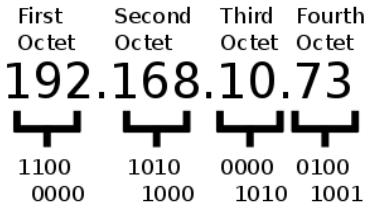


Figure 5.2: IP address construction

An IP address is simply a 32-digit binary number. In fact, that's exactly how computers interpret it. The IP address 192.168.10.73 is actually the binary number 11000000101010000000101001001001. To make it more readable, we convert each octet (eight bits) into it's decimal notation, and separate each octet with a period.

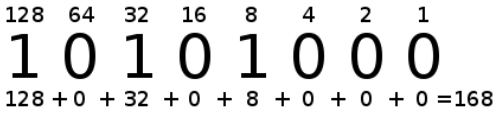


Figure 5.3: Converting from binary to decimal

When counting in binary, each digit can have a value of 0 or 1. To reach a value of 2, you need to use a second digit. To reach 4, you need a third digit. To reach 8, you need a fourth digit, and so on. You can represent the numbers 0 through 15 using four binary digits, or 0 through 255 using eight binary digits. Figure 5.2 describes this process.

You may have noticed that on your home network, or in the exercises we've done so far, the network we're on has a fixed first three octets, and the device is specified using the last octet. In this case, the first 24 bits (first three octets) make up the **network address**, and the remaining 8 bits make up the **host address**. This is a common convention. You frequently see such networks written like this: 192.168.10.0/24, meaning that the first 24 bits of the 192.168.10.0 address refer to the network address, and the remaining 8 bits can be used to identify hosts on that network.

Now, you may be thinking that with 8 bits to specify hosts, we can specify up to 256 hosts. After all, 2^8 is 256 and the possible values for that last octet are 0 through 255. However, in every network, there are two special addresses that are not used to identify hosts. The first special address is called the **broadcast address** and is the address where all of the host bits are set to 1's. In this example, it would be 192.168.10.255. The broadcast address is used to send a message to all hosts on a network. The second special address is the network address itself, where all of the host bits are set to 0's. In this case, that would be 192.168.10.0. So, subtracting those two special addresses gives us 254 usable host addresses on the 192.168.10.0/24 network.

Subnets

So far, we've only talked about the host side of the IP address, in a specific example with 24 network bits and 8 host bits. However, networking is hierarchical, and any network can be broken up into smaller networks. These smaller networks are called subnets. Actually, virtually any network can be considered a subnet, except for 0.0.0.0/0 which encompasses all possible networks.

As a concrete example, lets say we wanted to segregate our home network into two networks of equal size. Originally, our home network was at 192.168.10.0/24. Splitting it into two even networks would create the 192.168.10.0/25 network and the 192.168.10.128/25 network. Looking at it in binary may make it a bit clearer.

192.168.10.0/25 = 11000000.10101000.00001010.**0**/0000000

192.168.10.128/25 = 11000000.10101000.00001010.**1**/0000000

By splitting the network in half, one additional bit is used to represent the network address, and one fewer is used to represent the host address. This means that there are only 128 possible combination of host bits for each network, and after we subtract out the two special addresses, only 126 possible host addresses. We call these two new networks **subnets** because they are subsidiary networks of the original.

What if we split the original network into four networks?

192.168.10.0/26 = 11000000.10101000.00001010.**00**/000000

192.168.10.64/26 = 11000000.10101000.00001010.**01**/000000

192.168.10.128/26 = 11000000.10101000.00001010.**10**/000000

192.168.10.192/26 = 11000000.10101000.00001010.**11**/000000

Now we've used *two* bits that used to be host bits to specify the network address, and split what used to be the 192.168.10.0/24 network into four subnets, each with 64 IP addresses and 62 possible hosts.

The same logic works the other way as well. The 192.168.10.0/23 network looks like this:

192.168.10.0/23 = 11000000.10101000.0000101/**0**.00000000

We now have a network with 9 host bits, 512 possible IP addresses, and 510 possible hosts.

Note that to control the 192.168.10.0/23 network, we would have to control both the 192.168.10.0/24 and 192.168.11.0/24 networks.

We will use these subnetting skills to understand how routing works on the Internet in a later section.

Practice: Subnetting

1. Which subnet does host 192.168.24.153/27 belong to?
2. What is the first valid host on the 10.0.3.8/29 network?
3. How many valid hosts are there in the 172.16.16.0/26 network?
4. What is the broadcast address of the 10.22.8.0/22 network?

Assigning IP Addresses with DHCP

So, we know about IP addresses. We know which IP addresses are reserved for hosts. We know how to move around on the network, listen to network traffic, and figure out our own IP addresses. But how do machines get their IP addresses in the first place?

There are two ways. First, you can set your IP address manually. Second, you can use DHCP (Dynamic Host Configuration Protocol).

There are two parties in any DHCP handshake: The client and the server. The client is a computer that uses DHCP to get a new IP address from a reserved pool of IP addresses on the network. The server is a computer that responds to DHCP requests and hands out IP addresses to clients that ask for them.

The DHCP protocol has four phases, conveniently forming the acronym DORA:

Discover	The client sends a layer 2 broadcast request looking for a DHCP server
Offer	The server responds directly, supplying an IP address for the client
Request	The client formally requests that IP address
Acknowledge	The server registers that IP address to the client

Each of these phases takes one packet, so when you look at DHCP traffic, you will see a complete handshake in four packets.

Exercise: Set up a DHCP server

In this exercise, you will set up a DHCP server and then watch as two clients are granted IP addresses via DHCP. To start, navigate to `lesson5/exercise2/` and run `server.sh`. This will bring up the DHCP server. It will also set up some fun networking rules so that your DHCP server can get Internet access, assuming your host has Internet access. If not, this exercise won't work at all. Go find Internet.

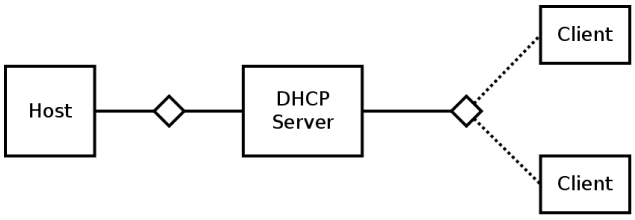


Figure 5.4: Exercise 2 network topology

Follow these instructions:

- `ssh` to the server at 192.168.10.2
- Notice that `ifconfig` shows we connected via 192.168.10.2 on `eth0`, so it's likely that the clients will show up on the network connected to `eth1`
- Manually set the IP address of `eth1` using `ifconfig eth1 192.168.20.1/24`
- Use `apt-get` to install `udhcpd`, a DHCP server package (don't forget to `update` first)
- Use the text editor `nano` to edit the configuration file, `/etc/udhcpd.conf`. This file is owned by `root` so be sure to use `sudo`.
    - Since our `eth1` is hosting the 192.168.20.0/24 network, lets hand out IP addresses in the range of 192.168.20.2 through 192.168.20.254
    - Set the `start` and `end` values appropriately

    - Change the `interface` value from `eth0` to `eth1` so we hand out DHCP addresses via the `eth1` interface
    - You can use <ctrl-x>, <y>, <Enter> to save and exit
- Enable the DHCP server by editing `/etc/default/udhcpd` (with sudo!) and commenting out the second line like so:
    - `#DHCPD_ENABLED="no"`
- Now, restart the DHCP server
    - `sudo /etc/init.d/udhcpd restart`

At this point, your DHCP server should be up and running. Start running `tcpdump` on the DHCP server on the interface the clients are going to connect to (which interface is this?) so that you'll be able to see their DHCP handshake in the next step.

Hint: Use the `-nn` option for `tcpdump` to make the output easier to read. Sometimes without using the `-nn` option, tcpdump will stall out and not display anything at all.

Open a second terminal or exit your `ssh` session to return the `lesson5/exercise2/` and run `clients.sh`. This will launch two clients on the same network as your DHCP server that will attempt to use DHCP to obtain an IP address.

In your `tcpdump` window, watch the DHCP handshake and/or ARP requests to determine which IP addresses were leased to the clients and verify that you can access them by using `ssh` to log in to them.

Hint: If your DHCP server isn't configured correctly, you can fix it and restart the clients by rerunning `clients.sh`.

Routing

So, at this point, we should be pretty familiar with IP addresses and comfortable using ssh to move about the network. In exercise 1, we even moved between computers on different networks. Now, let's discuss how network traffic travels between networks.

If you take a look ahead to Figure 5.5, you can see that each **router** is connected to multiple networks. The purpose of a router is to connect networks and make routing decisions. For instance, if Router0 gets a packet with source address 192.168.2.1 and destination address 192.168.4.2, the quickest and most efficient way to send that packet is through Router1. In most situations, it would be silly to send it to Router2 first, as Router2 has no connection to 192.168.4.2 and would have to forward it on to Router1 anyway.

So, that's basically what routing is all about: forwarding packets on different networks along efficient routes so hosts on one network can connect to hosts on another network.

Now, the trick is the whole *efficient* thing. How do routers know the best routes to send network traffic along so that it gets to its destination quickly and efficiently? They have a lookup table.

Again, looking ahead to the Figure 5.5 and the next exercise, Router0 needs to know where to send network traffic it receives. To do so, it uses a **routing table**.

Table 5.1: Example Router0 routing table

Destination	Next Hop
default	192.168.2.1
192.168.4.0/24	192.168.3.2
192.168.6.0/24	192.168.3.2
192.168.7.0/24	192.168.5.2

When Router0 gets a packet with a destination address of 192.168.7.2, it knows that none of its interfaces are directly connected to 192.168.7.0/24, so it looks up the row with a matching destination address, sees the next router to send it to is Router2 at 192.168.5.2, and forwards the packet to Router2. Router2, in turn, receives the packet destined to 192.168.7.2, sees that it is directly connected to 192.168.7.0/24, and sends the packet directly to Client1.

When Router0 gets a packet with a desination address of 192.168.3.2, it knows it has an interface already directly connected to the 192.168.3.0/24 network, and can send that packet directly to the recipient without needing to use the routing table.

In the upcoming exercise, all of the routers' routing tables are manually populated with routing rules. In real networks, routers almost always constantly reconfigure their routing tables based on varying network loads to try to find the most efficient route for each packet. Two of the common protocols in use today are OSPF (Open Shortest Path First) and BGP (Border Gateway Protocol). However, dynamic routing protocols are beyond the scope of this course. As long as you understand how a router uses its routing table to make routing decisions, you'll be fine.

Note: When a router goes down, backup routing rules will continue to route packets through an alternate, less-efficient route. This is why the Internet is so resilient!

Exercise: Break the Internet

In this exercise, we want to demonstrate how routing works. To start, navigate to `lesson5/exercise3/` and run `start.sh`. This will set up the network shown in Figure 5.5, a small "Internet" simulation.

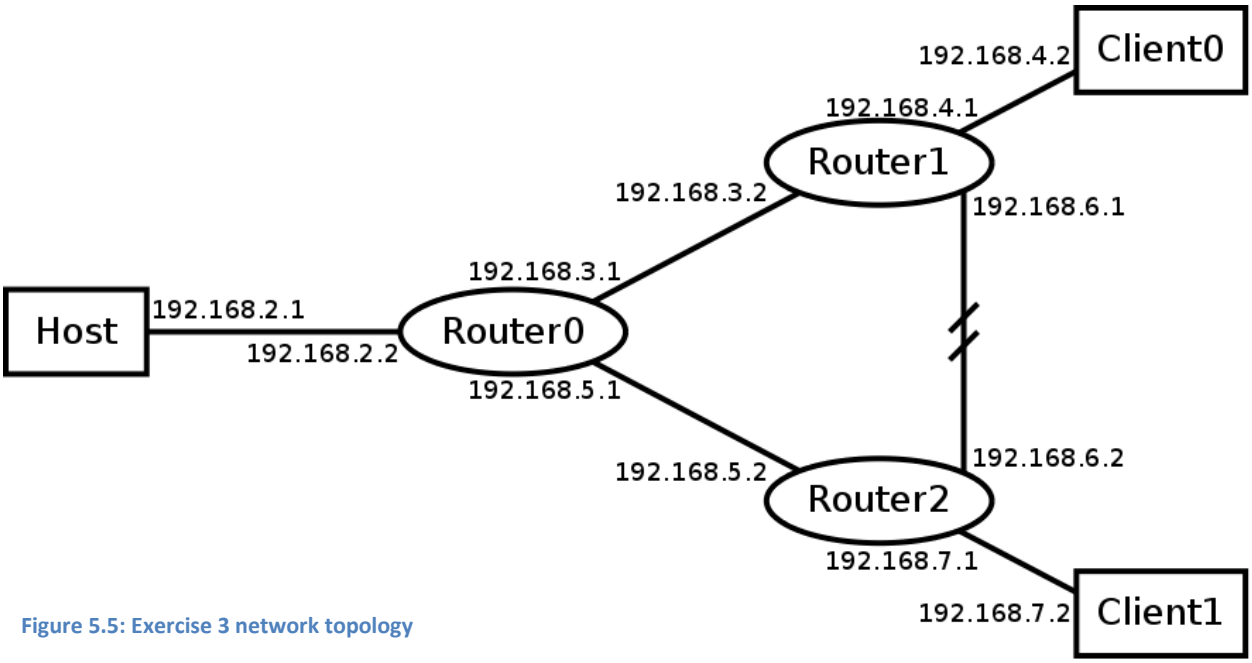


Figure 5.5: Exercise 3 network topology

Follow these instructions:

- `ssh` to Client0 and Client1 directly from the Host (pretty cool, huh?)
  - Hint: because the routers are configured to route traffic between the different networks, you can `ssh` directly from the host to any machine on the network, without having to `ssh` through intermediate machines.
- `ssh` to each of the routers and look at their routing tables.
  - `sudo route -n`
- `ssh` to each client and use `traceroute` to map the network path to the other client
- Once you get familiar with the network, run `exercise3/breaktheinternet.sh`
- `ssh` to a client and `traceroute` to the other one. What's different from before?
- Investigate the change in the network. What happened to the network? Why didn't the "Internet" break?

Lesson 6

In this lesson, we will cover Layer 4 of the OSI model, TCP and UDP, ports, NAT, and port forwarding. Let's start by discussing TCP and UDP.

TCP and UDP

Previously we discussed the OSI model, IP addresses, and how addressing works with the Internet Protocol (IP). IP, in the OSI model, takes place at the Network Layer (Layer 3). Remember that the Transport Layer is responsible for breaking up messages into manageable chunks and reassembling them later. By far, the two most common Transport Layer protocols are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). From here on in, we will just call them TCP and UDP.

Why is it important that we discuss these communication protocols? Communication protocols are important because they are the defined set of rules for exchanging information between computers. They give us the standards which we use to communicate, and protocol selection can be very important. Let's compare TCP and UDP.

TCP

TCP is a connection-oreinted protocol that was written to complement the Internet protocol, so you may also see TCP referred to as TCP/IP. Being a connection oriented protocol means that a connection is established before any useful data can be sent. No connection, no cool data.

Requiring a connection for TCP makes data transmission more reliable, but can increase latency and depending on the use case, that might be a big deal. TCP is used for most of your internet communication where reliably getting the entire message is important, such as surfing the web, sending and receiving email, and transferring files. You really *need* all the information in order for an email to make sense.

Example:
First two words dropped in a message - "ATTACK AT DAWN !!! "
The entire message – "DO NOT ATTACK AT DAWN !!!"

TCP establishes a connection by following this given procedure, which begins with a three-way handshake.

1. A server typically is listening for a connection request on an open TCP Port.
2. A client sends a **SYN**, asking to start the connection.
3. The server responds with a **SYN-ACK** with a matching connection request.
4. The client then, in turn, replies with an **ACK**. Congrats, you just learned the SYN-ACK handshake.
5. Now a connection is established, and the data can be transmitted.

Data is sent with each packet group acknowledged (ACKed) to ensure that it was received and does not need to be resent. The information to be able to do this is present in the packet, as all TCP packets are made up of a **header** and a **data** section. The header may contain the address information (source and

destination port), the sequence number (important to know if packets were received out of order or if part was missing), an acknowledgment number (so the sender can tell if a packet went missing), and a checksum (lets us know that the whole packet was received). There is more to the packet header than this, but this is what you need to know for right now. If data packets are sent and not acknowledged by the other side, they will be retransmitted.

Closing the connection is required once all of the data has been sent. This is the called the four-way handshake.

1. The client sends a **FIN**, indicating that they would like to close their half of the connection.
2. The server acknowledges this receipt with an **ACK**.
3. If the server or endpoint is ready to close their connection then they also respond with a **FIN**.
4. On receipt of the **FIN**, the client then responds with an **ACK** and the connection is closed.

Figure 6.1 shows TCP in action from establishing the connection, transmitting data, and then closing the closing the connection.

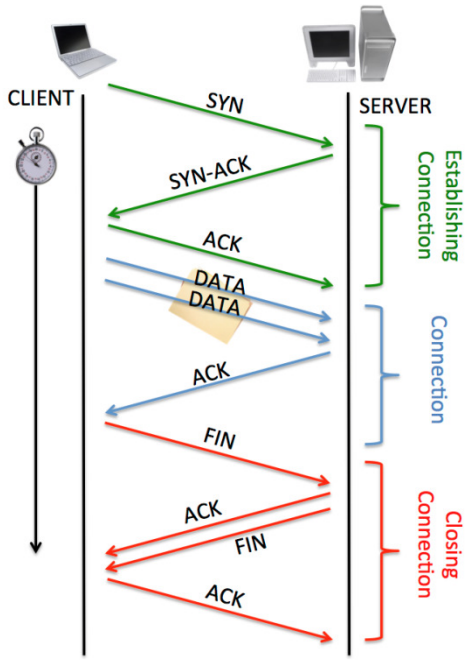


Figure 6.1: TCP protocol

You may notice that to send two packets of data, ten packets were sent. For small payloads requiring high reliability and assurance, this tradeoff may be worth it. For other small payloads requiring low latency, this can be a significant downside.

UDP

UDP is a connectionless protocol, which means that there are no handshakes taking place. This makes UDP much less reliable as there is no guarantee of delivery, ordering, or duplicate protection. If you

want these features in your application, you would need to build them yourself. This is frequently done. UDP does provide checksums to verify data received.

UDP does excel in a few key ways. Time sensitive applications use UDP when error checking and correction isn't necessary or important. This is because, in some cases, we care more about not waiting for packets. Instead we want to *stream* content as fast and as uninterrupted as possible. Examples include gaming (lag) and live-streaming video and audio.

Example:
You want to stream live sports and aren't worried if your resolution decreases if you miss a packet, but you would hate it if the game stopped for a minute while waiting for that one packet that got dropped. Unless it's the Justin Bieber halftime show … and then you want ALL the packets to get dropped.

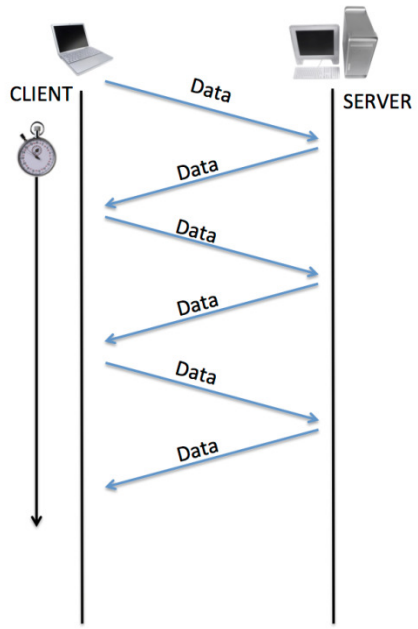


Figure 6.2:UDP protocol

UDP in action looks a lot simpler than TCP, but that simplicity comes at the cost of data reliability.

Ports

Let's think about what's happening now in the Transport Layer when we try to pass messages from the Transport Layer to the Application Layer. If you're using a chat program to send your friend a message, and your friend sends a response, one or more packets will be sent to your machine on the network. In layers 2 and 3, the packet is addressed and routed to your computer, but once it has reached your computer, the network stack needs to pass the packet to the particular application that is expecting it. If we were running two chat programs on the same computer, you wouldn't want the messages to be sent to the wrong chat program, right?

To this end, when an Application begins to listen on the network for incoming packets, it registers a special number called a **port** with the Transport Layer of the networking stack. Then, when incoming

packets come in destined for that port, the Transport Layer knows to which application to deliver the message. So that there can never be a conflict, only one application can register each port. The possible ports are 0 through 65535, although the first 1024 ports are reserved for special services.

Both TCP and UDP protocols use this notion of ports, although one port number can be registered for TCP and UDP at the same time. They do not conflict.

Netcat

We are going to need to explore some tools in order to view and learn more about these connections. The first is `netcat`. Pro Tip: You will also see netcat run as `nc`.

`netcat` is considered the swiss army knife of networking tools. It can be used to monitor, test, or send data across networking connections. The most basic syntax for using netcat is:

```
netcat [options] <host> <port>
```

Deceptively simple, right? For more information remember that you can run `--help` after the command. In order to connect to a TCP server, try the following.

```
netcat <ip_address> <port>
```

To connect to a UDP port you will need to add the `-u` option.

```
netcat -u <ip_address> <port>
```

Finally, you might want to know how to set up a server to listen for incoming traffic:

```
netcat -l <port>
```

Or, using UDP:

```
netcat -u -l <port>
```

And to redirect that traffic to a file:

```
netcat -l <port> > <filename.txt>
```

Exercise: Make a network connection with netcat

Navigate to `lesson6/exercise1` and run `start.sh`. This will set up a single virtual machine listening at 192.168.10.2 on TCP port 4321 and UDP port 4321. Connect to both ports using netcat and send the server some data (text).

Over TCP port 4321, what is the server doing to your data?

Over UDP port 4321, what is the server doing to your data?

Ports in use

What if we don't know what ports are open for communication? Then we need to use another tool to look things up for us. `netstat`, or network statistics, is a command line tool that displays network connections for TCP, routing tables, and both interface and protocol information. To get a list of ports in use, try the following:

```
netstat -a -n -t
```

`-a` prints out both active and listening ports in use (by default, just active ports)
`-n` makes netstat not try to resolve ports into names
`-t` only displays TCP ports in use

You can also try `-u` to display UDP ports in use, and you can condense all the options like this:

```
netstat -ant
```

```
netstat -anu
```

What information were you able to get about ports and connections using the netstat command?

Exercise: Infiltrate the DeathStar

Navigate to `lesson6/exercise2` and run `start.sh`.

In this exercise, your goal is to infiltrate the VM known as the DeathStar and steal its Master Key.

Unfortunately, we don't know where the Master Key is hidden. You will have to use the skills you've learned so far in this class to find it.

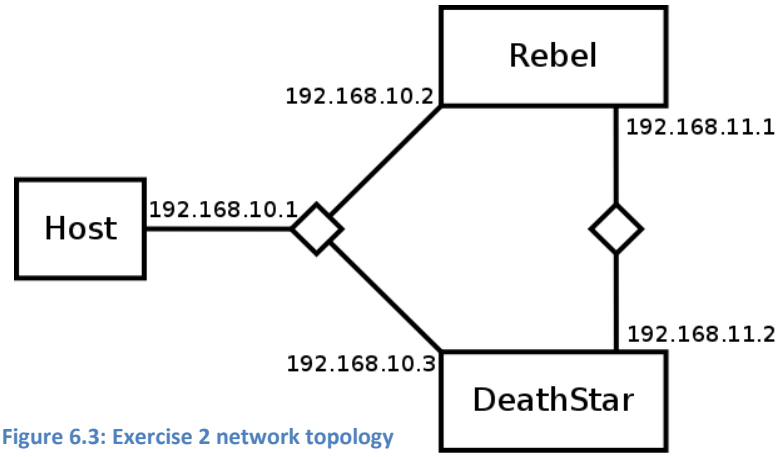


Figure 6.3: Exercise 2 network topology

We do know that the DeathStar has enabled an advanced network defense capability called a **firewall**. A firewall allows a machine to block network traffic based on its protocol (TCP or UDP), port number, or even its source or destination IP address. You will learn about firewalls in Lesson 9.

Hint: When attempting to connect to the DeathStar, you may be able to evade the firewall by connecting via a different route.

Hint2: The DeathStar may have ports open and listening. Try to figure out what these are, as it's most likely to have openings in its firewall for those ports. Not every listening port will necessarily be

accessible. Ports below 1024 are typically reserved for the system. Ports above 1024 are probably more interesting.

Hint3: Leaving a second terminal logged in on the DeathStar may give valuable intel as status updates are broadcast to the ship.

Network Address Translation (NAT)

Let's do a little math. Possible IP addresses range from 0.0.0.0 to 255.255.255.255, or alternately, there are 2^{32} = 4.3 billion possible IP addresses. There are a lot more computers in the world than that (remember, phones are computers these days, your thermostat might be a computer, etc.), so how do we deal with giving each device a unique IP address?

Trick question – we don't.

While originally, the Internet may have been expected to give each device a unique IP address, we quickly ran out of addresses to allocate, so several **private IP address bands** were allocated for private, overlapping use. One of these is the 192.168.0.0/16 network, which is why most people's home networks use that IP space. One restriction on the private IP space is that because it exists in so many locations, Internet-facing routers can't possibly route packets with private source or destination addresses. Consequently, any packet transmitted on the Internet with a private IP address will be immediately dropped.

So, if our home networks all use private IP address space, how is it that we can reach Google just fine? Well, that's the magic of Network Address Translation, or NAT.

NAT is a way to re-map one IP address space to another by modifying network address information in IP packet headers while they are in transit. That is, when your computer sends a packet to `google.com`, the packet's source address is set to 192.168.1.15 (let's say that's your computer's IP address), and your computer passes it to your home network's router for distribution out into the Internet. Your home router has a NAT running that remaps the packet's source address to the router's own *external* address. This makes more sense with a picture, so take a look at Figure 6.4.

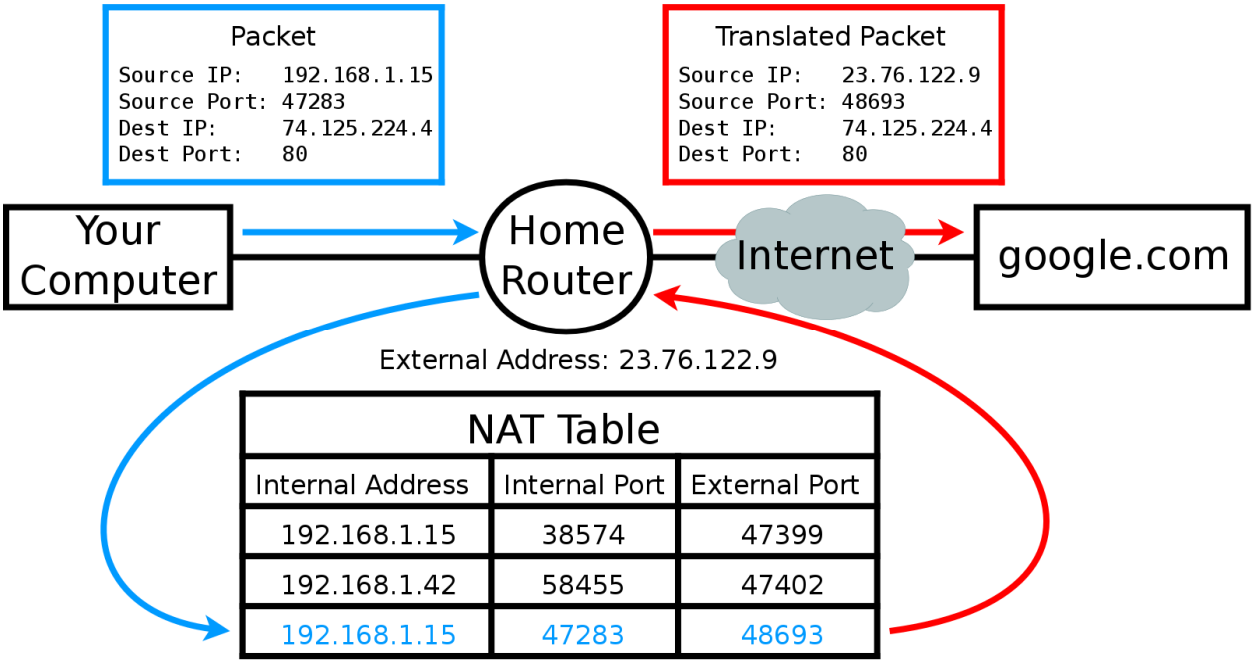


Figure 6.4: Network Address Translation

When you send a packet to `google.com`, your packet will have your computer's source IP and port number, and Google's destination IP and port number. When you send this to your home router, it knows that it is supposed to translate outgoing packets so they appear to originate from the router, so it changes the source IP and port to its own IP address, 23.76.122.9, and a *unique* high-numbered port, 48693. It records this mapping in its NAT table.

When `google.com` responds, it will send a response to 23.76.122.9 port 48693. Your home router will receive this packet, look up the internal address and port mapped to external port 48693, and re-translate the packet to have a destination address of 192.168.1.15 port 47283. Your computer will receive the packet and have no idea the packets were translated at all.

Exercise: Look behind the NAT

Navigate to `lesson6/exercise3` and run `start.sh`.

In this exercise, the web server running on your host is being accessed by one of the virtual machines every 5 seconds.

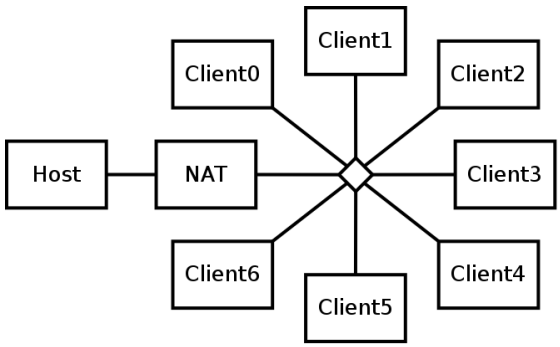


Figure 6.5: Exercise 3 network topology

Which machine and port does the Host machine see making the requests?

Which machine is actually making the HTTP requests?

Port Forwarding

One of the side effects of using a NAT is that the machines behind the NAT are difficult to reach from outside the local network. If `google.com` randomly decided to send a packet to Your Computer, it would reach your home router, your home router would look up the destination port in its NAT table and… there wouldn't be an entry there. Your home router wouldn't know where to send the packet. Bye bye packet.

Good thing there's a work-around. If you want to expose a service running on your computer to the Internet, you can set up a static rule to forward all traffic on a particular port to a particular host on the local network. For instance, let's say your computer was running a web server on port 80 that you wanted to grant access to. You can configure your router to listen to port 80 on its external interface and forward (translate) all packets sent to that port on to your computer.

However, since the router only has one port 80 on its external interface, you can only forward port 80 to a single machine on your local network. If you wanted to expose a two web servers on your local network, you would need to pick another port to expose the second one on.

Guide: Enabling port forwarding on a home router

By default, all home routers come with NAT enabled and all inbound access to the home network blocked. However, there are times when you may want to enable inbound access to your home network on some particular port. You may be running an SSH server and want access from a friend's house. You could be hosting your own website. Both of these require you to set up port forwarding to allow inbound access to your network from the Internet.

Warning: Opening access to your home network from the Internet is *very* risky. Perform the following at home *only* if you are sure about the risk you are taking and are willing to assume that risk. People (both good and bad people) regularly scan the entire Internet for listening computers. One of the quickest ways to become part of a botnet is to improperly open a port and inadvertently make a vulnerable service accessible from the Internet.

To follow along without modifying any actual router configuration, open a web browser and go to http://voiproblem.com/emulators/Netgear/WGR614v6/start.html for a simulated router configuration page. Note: your home router configuration page will likely look different, depending on the particular brand and model of your router.

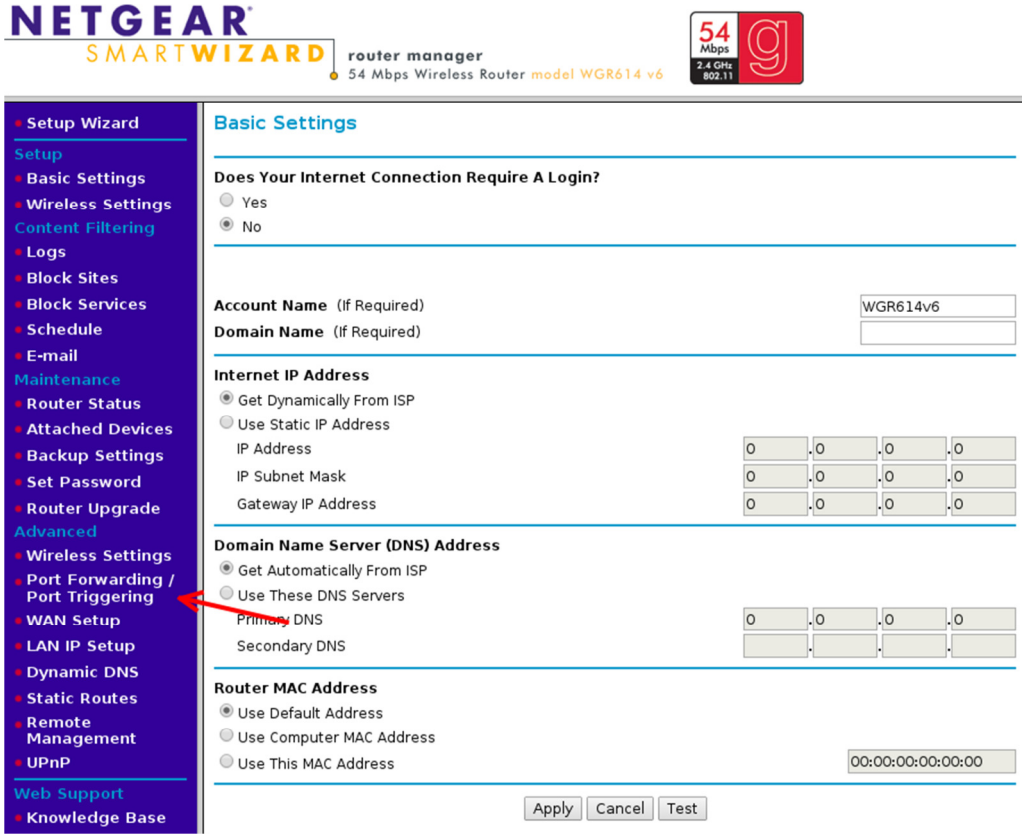


Figure 6.6: Netgear router main page

To start, navigate to your router's admin page, which is typically just hosted on TCP port 80 at your router's IP address. This is usually 192.168.1.1, but if for some reason it is different on your network, you can use `tcpdump` to look for DHCP, ARP, or other traffic, or use `route` to see your default gateway. You will probably have to log in. If you've never set the username and password before, you can use Google to find the defaults.

When you've gained access to your router's web interface, click around until you find Port Forwarding. Unless you're following along with the fake router configuration page, it will probably look different on your router.

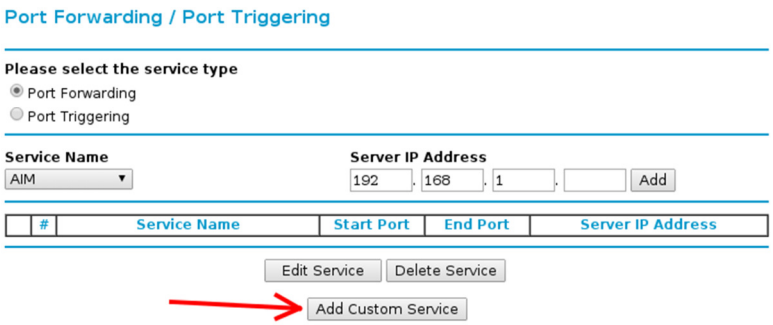


Figure 6.7: Port forwarding

Next, you'll want to make a new rule. Some routers will come populated with pre-defined rules you can select, but since you now know about TCP, UDP, and port numbers, it's easier to just create a custom rule.

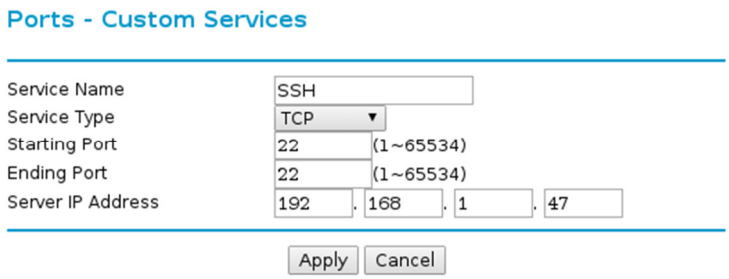


Figure 6.8: Adding a custom rule

Finally, we're ready to create our custom port forwarding rule. In this example, we're going to forward TCP port 22 (and no other ports) from the external (WAN) interface of the router to IP address 192.168.1.47 inside our home network. Presumably the machine at 192.168.1.47 has an SSH server listening on port 22 for new connections.

After you hit Apply, any new connection attempt to connect to the router's external IP address on port 22 will be forwarded on to the machine at 192.168.1.47. When it responds, the router will mangle the outgoing packets to make the source IP address appear to be the router's external IP address.

Now you can access your home network from anywhere!

Lesson 7

In this lesson, you will learn about the Application layer(s) of the OSI model.

HTTP Servers

One of the most ubiquitous activities these days is accessing web pages through the Internet. How this works is really quite simple, but fundamental to how the Internet is constructed.

By convention, HTTP Servers, or Web Servers, run on TCP port 80. That is, if you take any computer and run a web server application on TCP port 80, you are hosting a website. By default, your website might not be terribly interesting, but it will exist.

Go ahead and check it out by opening a web browser and navigating to `localhost` in the address bar.

You should see a webpage pop up, possibly saying something along the lines of "It Works!" or "Default Page". What's happening here is that `localhost` maps to 127.0.0.1, which is a built-in IP address on each computer referring to itself, so the web browser is connecting to the web server hosted on your own computer. The web browser also knows to connect to TCP port 80 by default to retrieve a webpage. If for some reason your web server were configured to listen on port 79, or 8000, you could access it by navigating to `localhost:79` or `localhost:8000`, respectively, in the web browser's address bar.

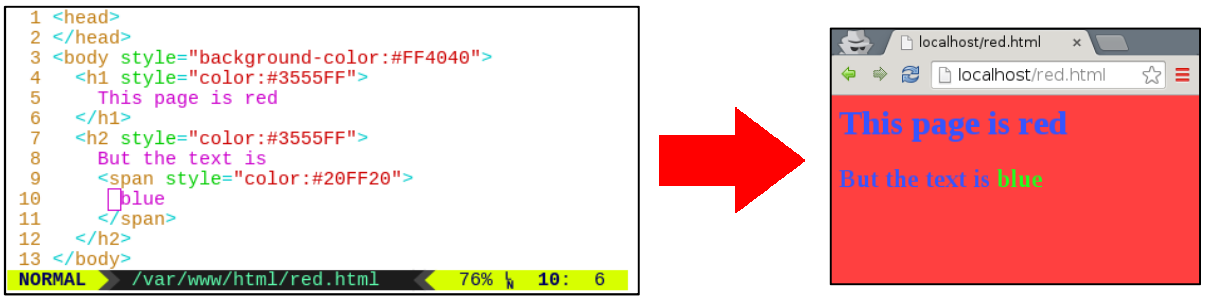


Figure 7.1: Turning HTML into a web page

What is actually happening is that your web browser is simply retrieving a file from the server using the HTTP protocol, then reading and *rendering* that file into a visual webpage. For instance, in Figure 7.1, our web server is hosting an HTML file at `/var/www/html/red.html`. When we navigate to `localhost/red.html`, the web browser retrieves this text file and follows the formatting instructions in the HTML code to create a graphical representation of the website.

In more complicated webpages, the HTML code may import additional content into the page using special tags such as `<img>` and `<link>`. Once the initial page (say, red.html) has been retrieved from the web server, the browser will read it and then go retrieve the additional page elements. See Figure 7.3 for an example of the Chrome browser loading the web page shown in Figure 7.2.

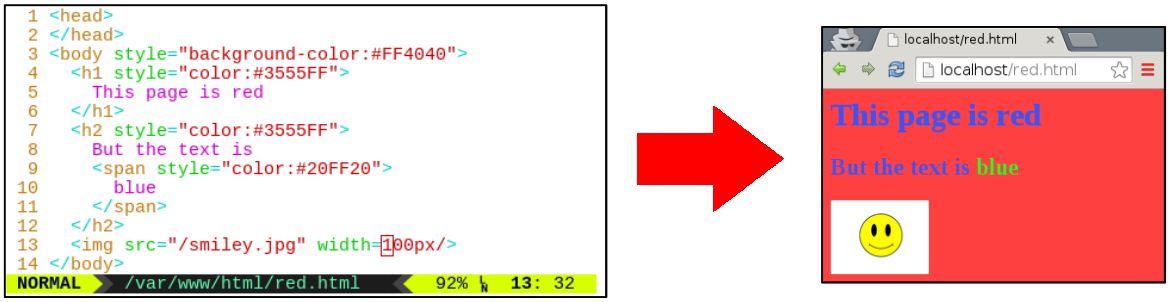


Figure 7.2: Loading additional page elements

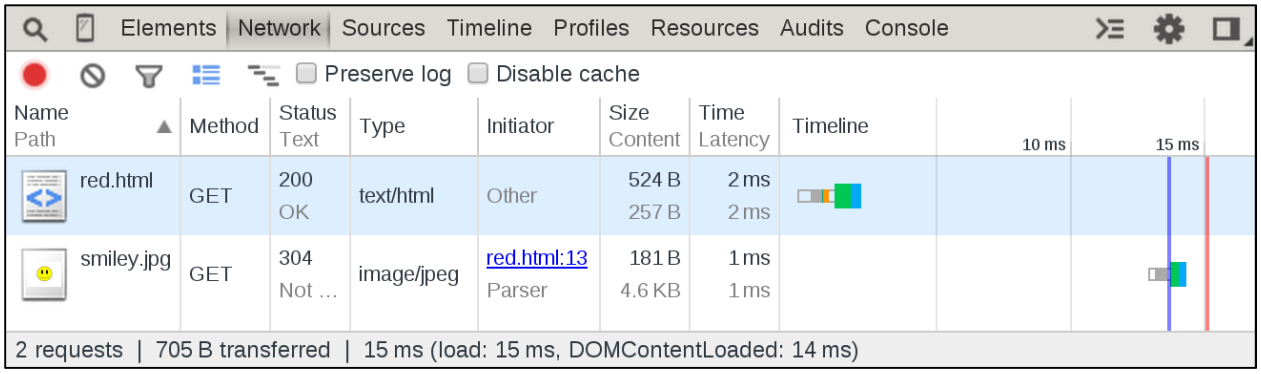


Figure 7.3: Chrome retrieval of page and page elements

Exercise: Building a website

In this exercise, you will configure and run a web server to host a website you build. Then, you will construct a website by pulling content from three partially-constructed sites available on the virtual network.

Part 1:

Your computer should already be running an apache2 web server on port 80 (it does this when you install apache2, which we already did for you).

- Open up a web browser and navigate to `localhost` (type “localhost” into the browser address bar)
  - By default, web browsers will try to connect to web servers on TCP port 80 if no port is specified
  - You should see the debian default web server page with the text “It works!” near the top
- Now, try to access `localhost:8000`
  - You should get a message saying “This webpage is not available” or something similar

Why?

We want the web server to listen on port 80 (which it does by default) and also on port 8000 (which it does not).

- Open up `/etc/apache2/ports.conf` for editing (you will need to be root)
  - Under the line `Listen 80`, add another line `Listen 8000`
  - This will make apache2 listen for connections on port 8000, but we haven't yet configured what we want it to do with those connections
- Open up `/etc/apache2/sites-enabled/000-default.conf` for editing (as root)
  - You will see an XML file with a bunch of options inside `<VirtualHost>` tags
  - Add the following to the bottom of this file:

```
<VirtualHost *:8000>
  DocumentRoot /var/www/html
</VirtualHost>
```

  - This will tell apache2 to serve files from `/var/www/html/` to connections on port 8000 the same way it is doing for connections on port 80
- Restart the apache2 web server
  - `sudo /etc/init.d/apache2 restart`
- Open your browser again and this time navigate to `localhost:8000`
  - You should see the same default web page
  - If not, go back and check that your configuration files are correct

Part 2:

To start, navigate to `lesson7/exercise1/` and run `start.sh`. Three new virtual machines will launch, each running a web server on port 80 at 192.168.10.2, 192.168.10.3, and 192.168.10.4. Check out each website by entering each IP address into your browser's address bar.

Remember that our website is hosting files out of the directory `/var/www/html/`. If you take a look at the directory contents, you'll see a single file `index.html`. Go ahead and take a brief look at the file contents and you'll see a big jumble of HTML and CSS code. You're not expected to understand how it all works (this isn't a web design class), but you might notice that the text in `index.html` matches the text you see when your web browser loads the page at `localhost`. This is because the web server will serve whatever `index.html` file it finds in the base web root (`/var/www/html/`) by default.

- If you navigate to `file:///var/www/html/index.html` in your browser, you can see that the same page loads

The web browser takes an HTML file and renders its contents (frequently with help from Cascading Style Sheets, or CSS) for viewing in the web browser.

There are three main components to most websites: HTML, CSS, and images. Coincidentally, each of the three virtual machines you started contain one of these components and your job is to combine them into a fully-fledged website.

- 192.168.10.2 has the HTML you need to grab. The easiest way to do so is to use the web browser to load the page, then right-click anywhere on the page and select "View Page Source."

    - Copy the page source code, exactly as it is written, into `index.html`, overwriting its original contents.
- 192.168.10.3 has the CSS you need to grab. Unfortunately, our previous trick of grabbing the page source won't work for CSS files because they are loaded as a supplemental resource. However, if you look at the page source, you will see that it `@import`'s a CSS file named `style.css`.
    - Navigate to `192.168.10.3/style.css` to see the CSS source code.
    - Copy this into a new file, `style.css` into your base web root (`/var/www/html/`).
- 192.168.10.4 has the images you need to grab. This may be the simplest part. Simply right-click on each image and save it to `/var/www/html/`.
    - Be sure to leave the names the same; e.g. `top.jpg` should still be named `top.jpg` when you're finished.

Check out your new website by navigating to `localhost` or `localhost:8000`. If someone else is on the same network as you are (for instance, in a classroom environment), you can ask them to check out your website too by navigating to your IP address.

Domain Name System

Ever wonder what happens when you visit `google.com` and it magically displays their website? Well, we just worked through how the browser connects to a server and retrieves a web page, but how does your browser know where to find the server hosting `google.com`? That's the magic of DNS!

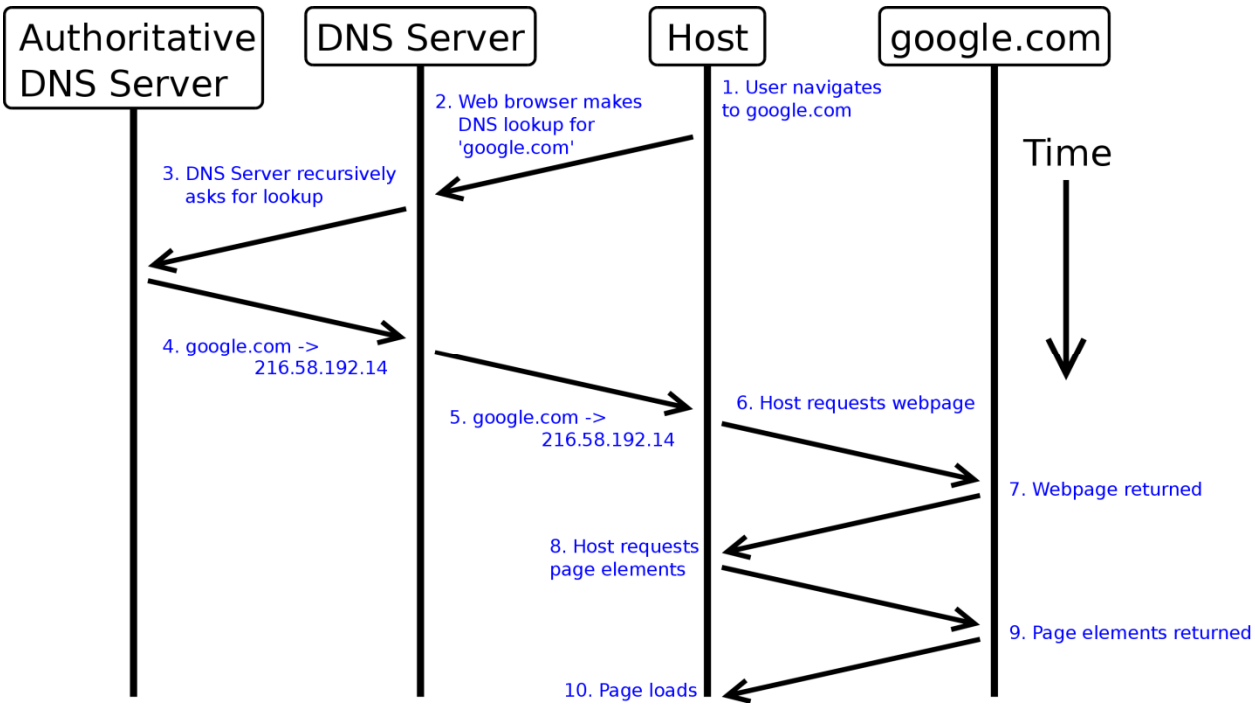


Figure 7.4: DNS Lookup

DNS stands for Domain Name System, and it's a system that maps each domain name to an IP address.

To do so, your host must be configured to find its DNS server, or *nameserver*, by its IP address. After all, if it only knew the nameserver's hostname, how would it resolve the hostname to an IP address?

Your host will send a DNS request to its nameserver asking for the record associated with `google.com` and the nameserver will send a DNS response with the mapping between `google.com` and IP address 216.58.192.14 (although google.com actually maps to many IP addresses; you can try it!). If the nameserver doesn't currently have a record for `google.com` in its cache, it will itself make a DNS request to another nameserver, or perhaps the *authoritative nameserver*, the nameserver principally responsible for the record for `google.com`.

The hosts file

DNS tends to work very well for websites on the Internet, but often in small projects, setting up a private DNS infrastructure is time-consuming and costly. In these cases, often manually configuring a mapping between hostnames and IP addresses in the `hosts` file is sufficient. On Linux this file is located at `/etc/hosts` and on Windows it is frequently located at `c:\Windows\System32\Drivers\etc\hosts`. When looking up a hostname, computers will check the hosts file, and only if it doesn't have an entry for that hostname, send a DNS request to look up the hostname.

Exercise: Resolve Domain Names

To start, navigate to `lesson7/exercise2/` and run `start.sh`. Three new virtual machines will launch as shown in Figure 7.5.

In this exercise, you will be helping two virtual machines resolve your computer's new domain name so they can access your website. Both of them are attempting to access `bestwebsiteever.ict.cta`, but are unable to resolve the domain name to an IP address.

For Client0, your goal is to use the `/etc/hosts` file to allow it to resolve the hostname to the Host's IP address.

For Client1, unfortunately the `student` user doesn't have root access, so it won't be possible to modify `/etc/hosts`.

However, if you take a look at Client1's `/etc/resolv.conf` file, you can see that it will use 192.168.10.2 as its nameserver. Log in to the DNS Server and edit its configuration as follows:

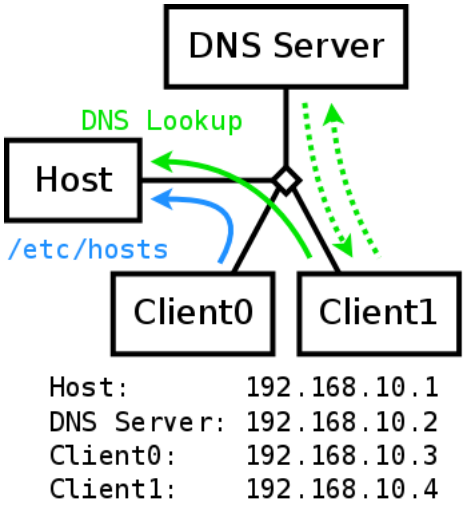


Figure 7.5: Exercise 2 network topology

- Navigate to `/etc/bind/`
- Edit the forward-resolution configuration file, `forward.ict.cta`
    - Near the bottom of the file, add a new entry for `bestwebsiteever` and associate it with IP `192.168.10.1`.
    - Use the entries for Client0 and Client1 as a reference.

- Edit the reverse-resolution configuration file, `reverse.ict.cta`.
    - Near the bottom of the file, add a new entry for `1`, with a PTR record to `bestwebsiteever.ict.cta`.
    - Use the entries for Client0 and Client1 as a reference.
- Reboot the Bind9 DNS Server
    - `sudo /etc/init.d/bind9 restart`

Verify that both Client0 and Client1 are making network requests to 192.168.10.1 by running `tcpdump` or `wireshark` on the host.

Lesson 8

In this lesson, we are going to cover a primer on data security through cryptography, some more advanced system administration tools and techniques, and how to monitor your system for intruders.

Sending secret messages

Cryptography is the study of sending and receiving secret messages, and it's a huge topic. If you want to learn more about it, we recommend taking the CTA class on Cryptography. However, in this lesson, we'll cover a few basic concepts.

Encryption

There are many instances where you might want to send a secret message to someone. You may not want your medical records transmitted in a way someone could snoop on them. You might be transmitting credit card or social security numbers and want to be sure that even if somebody is trying to listen in, they won't be able to capture that sensitive information. You can do this by using cryptography to **encrypt** your message into *ciphertext* before sending it, and allowing the recipient to **decrypt** the ciphertext back into the original message.

Caesar's Cipher

To give a concrete example of this, let's take a look at one of the classics: Caesar's Cipher. In this scheme, you can take your message M ("My name is Jenni"), and apply the following encryption routine to it: Rotate every letter by N places. For instance, if N is 1, then every "a" will become "b", every "b" will become "c", and so on.

```
N=0:  My name is Jenni
N=1:  Nz obnf jt Kfooj
N=2:  Oa pcog ku Lgppk
…
N=25: Lx mzld hr Idmmh
```

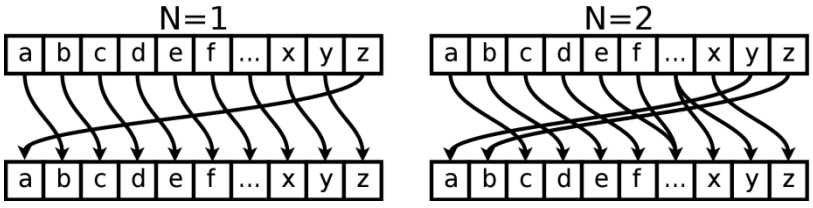


Figure 8.1: Caesar cipher

At first glance, this seems like a very effective method of making your messages indecipherable. You and a friend could very easily use a Caesar cipher to encode messages passed in class by picking a shared value for N. If the messages were intercepted by a teacher (the **adversary**), the teacher would most likely not be able to decode the message on the fly to embarrassingly read the message to the class.

However, when we're dealing with computers, everything changes. A computer could very quckly compute all 26 possible permutations of a Caesar cipher-encoded message. Instead, in most applications, we need an encryption scheme that protects secret data even if the adversary has massive amount of computing power.

There are several ciphers that rely on mathematical operations to encrypt and decrypt data that are simple to compute with a secret key, but often *very* difficult to reverse engineer without the key. A few examples are: RC4, AES, RSA, and ECDSA. Learning the specifics of how these cipher algorithms work is left as an exercise to the reader.

Authentication

Cryptography is also used all the time to perform authentication. Authentication, in a nutshell, is the process of verifying that you are talking with the entity with which you think you're talking.

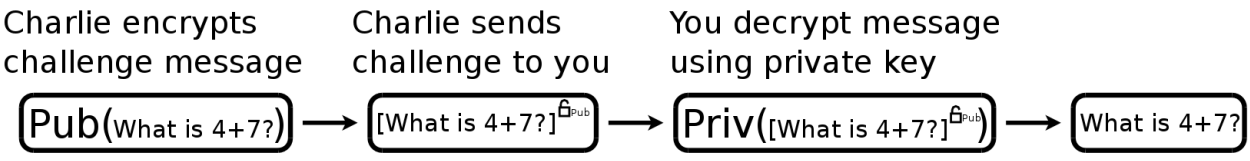
We do this all the time without generally thinking about it. Imagine a scenario where you're buying a new cell phone from a wireless carrier, and they need the social security number of the primary account holder (your mom) to put the new phone on the account. You might call up your mom and ask her for her social security number over the phone. Prior to giving it to you, she might check for some clues to know who it is she's talking with and that the situation is legitimate before disclosing her social security number. Does it sound like your voice over the phone? Did she know you were buying a new cell phone? Did caller ID show that it was your number calling hers? If any of these don't check out, she probably won't disclose her social security number (we hope).

A similar thing happens all the time on the Internet. Say you want to log in to your Google account. You might navigate to http://google.com in a web browser, which then redirects you to https://www.google.com. If your browser says "http**s**://" and doesn't pop up a big "WARNING" banner, then congratulations, you're almost certainly talking with Google. If http://google.com loads without the "s" in "http**s**://", or if you see a warning banner, there's a very real possibility that your connection is being intercepted and someone is trying to steal your account. The "s" in HTTPS stands for *secure*, and besides encrypting the content of the web page, it also uses **public key cryptography** to authenticate Google's server to you, so you know you're talking to the real Google.

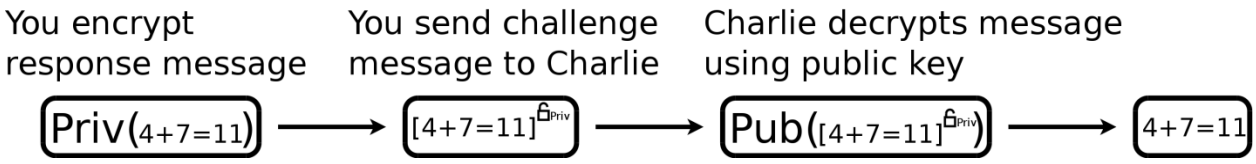
In public key cryptography schemes, if you want to be able to prove your identity to someone else, you will generate a pair of special keys. One is the *public* key and the other is the *private* key. As described in the name, the public key you want to distribute to the public. That is, you don't need to worry about keeping the public key secret. You want anyone to be able to find your public key if they need to verify your identity. On the other hand, your private key is *very* secret. Anyone who gains access to your private key can impersonate you, so it is critically important that you protect access to it.

The way these keys work is that either key can encrypt a message and that encrypted message can only be decrypted using the other key. So, a message encrypted with the public key can only be decrypted using the private key, and a message encrypted with the private key can only be decrypted using the public key.

To verify your identity, Charlie might send you a challenge message, "What is 4+7?" encrypted with your public key (he has your public key because it's *public*, remember).



Then, you would use your private key to decrypt the message, solve the challenge, "4 + 7 is 11", then encrypt the answer using the private key and send it back to Charlie.



Charlie would decrypt your coded message using your public key, verify the answer, and would therefore know that he was talking with you, the owner of your private key.

Of course, this entire authentication scenario is predicated upon Charlie being able to find *your* public key and attribute it to you, and there are a number of other mechanisms in practice to give some assurance of that. However, we aren't going to go into any of that here.

Exercise: Telnet vs SSH

In this exercise, you will explore the difference in security between Telnet and SSH.

Navigate to `lesson8/exercise1/` and run `start.sh`.

Then, open up `wireshark` and start capturing traffic on `veth0`.

Next, use `telnet` to log in to the server at 192.168.10.2

- `telnet 192.168.10.2`
  - `login: student`
  - `Password: password`

Note: the `student` user's password has been changed to `password` for this exercise.

As you type each character into the login and password fields, notice that a packet is sent and received.

Can you see what is in each packet? What are the security ramifications of this?

Next, use `ssh` to log in to the server at 192.168.10.2

- `ssh 192.168.10.2`
  - `Password: password`

As you type each character into the login and password fields, what do you see? What happens when you press enter? What useful information can you gather from the ssh packets?

SSH Tunneling

So, now that we've learned about encryption and authentication, why they're important, and that SSH uses both, wouldn't it be cool if we could leverage SSH to do other things with encryption and authentication? Hm… was that a leading question or what? Of course we can!

SSH is an incredibly powerful tool that not only allows you to log in and run comments on a remote system, but also lets you redirect, or **tunnel**, network traffic between hosts. Using the authenticated, encrypted TCP stream you've set up with the SSH server, you can funnel other network traffic through

that connection and redirect it to *come from* the SSH server. Let's look at an example to really explain what's going on.

Say you work for a company and they've got a restricted network environment. Probably, they will have an internal network that isn't generally accessible from the Internet. Say you're working on travel and you need to access one of your company's internal web pages. Most people at your company would be blocked – too bad. You, however, are a computer scientist, and computers exist to serve your needs, not the other way around.

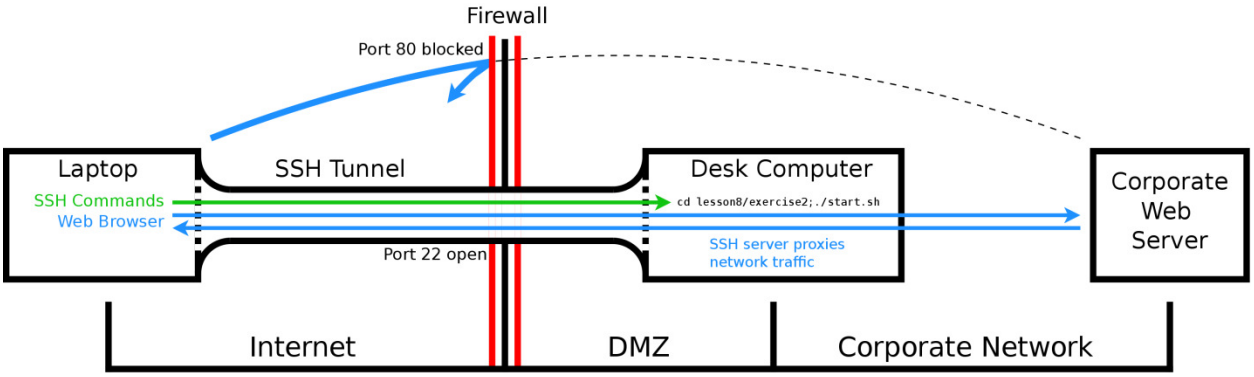


Figure 8.2: SSH tunnel

In complete compliance with your company's policy (make sure!), you have set up an Internet-facing SSH server to allow you secure remote access to your desk computer. Normally, you access your desk computer with the command:

```
ssh desk.company.com
```

But today, you also need to access the corporate webserver at `webserver.company.com`, so we're going to set up a *port forward* rule to forward traffic. Essentially, you specify a local port that the SSH client will listen to on your laptop, and any traffic sent to it will be encapsulated in the SSH tunnel and pop out at your desk computer destined for the specified host and port.

```
ssh -L <local port>:<remote host>:<remote port> desk.company.com
```

So, since we want to access `webserver.company.com` on port 80 (remember, web servers run on port 80), our command becomes:

```
ssh -L <local port>:webserver.company.com:80 desk.company.com
```

But what should we use for a local port? I don't know, pick something! Specifically, you probably want to avoid the first 1024 reserved ports, or common ports used by any services you know and care about to avoid conflicts, but if you pick a big number less than or equal to 2^{16}-1, or 65535, you should be fine. In this example, we'll just pick port 4444 out of thin air for convenience in typing (and that's usually a good enough reason).

```
ssh -L 4444:webserver.company.com:80 desk.company.com
```

Great! Now we've got a tunnel. If you opened a browser to localhost:4444, you would see the company web page.

Exercise: Tunnel an HTTP connection

In this exercise, you will learn how to tunnel network traffic through SSH.

Navigate to `lesson8/exercise2/` and run `start.sh`.

VM1 is hosting a webpage (TCP port 80) you wish to access, but unfortunately, VM0 is not configured to route traffic directly to VM1 for you. However, you do have SSH access to VM0, a machine that shares a network with VM1.

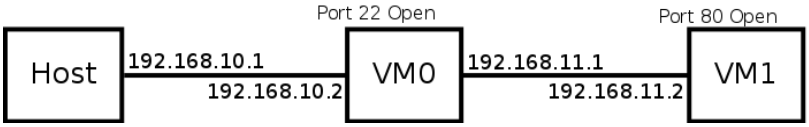


Figure 8.3: Exercise 2 topology

Open your web browser and try to go to "192.168.11.2". Notice that you will be unable to access that address.

SSH to VM0. From here, you can use `ifconfig` to see that `eth1` is on the 192.168.11.0/24 network, and can talk with VM1. Use the command-line tool `wget` to retrieve the webpage from VM0.

- `wget 192.168.11.2`

You will see that it downloads a file named `index.html`. Just like in lesson 7, this is the html for the website hosted at 192.168.11.2. One option is that we can copy this file back to the host machine, then view it there.

However, for this exercise, we want to use our web browser on the host to connect directly to the page hosted on VM1. Set up an SSH tunnel to VM0 to listen on TCP port 8000 on the host, and redirect that traffic to VM1 on port 80. Then, use your web browser to navigate to localhost:8000.

Logging Remote Logins

SSH is an incredibly amazing and powerful tool, but with all that power comes a risk – that someone else might get access to your server and be able to do everything you can. It's very common for bad people on the Internet to attempt to repeatedly attempt to log in to Internet-facing SSH servers in case you haven't changed the default username and password for the system. Some attackers will try to actively guess your password using brute-force methods or wordlists.

Because of this threat, it is a good idea to periodically check to see who has been trying to log in to your system and check for unauthorized activity. If someone is trying to break in to your computer, it will be very apparent because your log files will be filled with failed attempts.

If you detect an attacker, you might want to set up a firewall to block their IP address so they can't attempt to log in anymore. We cover firewalls in Lesson 9.

Finally, as an exercise for the reader, it is a good idea to use *SSH user keys* to log in to SSH servers instead of a username and password. Then, an attacker would have to try to break (or steal) your key to log in to your server. Much harder. Try this link: http://lmgtfy.com/?q=ssh+user+keys

Exercise: Authentication Failure

In this exercise, you will learn how to check who's been attempting to log in to your system.

Navigate to `lesson8/exercise3/` and run `start.sh`.

A VM will launch and repeatedly attempt to log into the host, simulating someone trying to brute-force guess a password.

You can find system logs detailing which users are trying to authenticate at `/var/log/auth.log`.

What username is being used to attempt to log in via ssh? Did the attacker successfully log in?

Lesson 9

In this lesson, we will cover more advanced Linux, networking, and system administration skills. We're getting to the really fun stuff.

Scripting

Often, you will find yourself needing to perform a series of actions repeatedly. For instance, you might want to keep a running tally of how many times someone fails to log in to your server each day. You could log in yourself every day and run a series of commands to get the number, or you could write a script to automate the process. Once you have a script, you can use another system utility called Cron to run your script automatically each day.

At its core, a script is just a text file that lists a series of commands in the order they should be executed. There are different kinds of scripts. There are shell scripts, that run shell commands, and there are other scripting languages, such as Python, that run Python commands. In this lesson, we are going to focus on shell scripts, specifically bash scripts, but if you take the CTA's Programming class, you will learn all about programming in Python, which can be much more powerful.

To start, lets look at a script that counts the number of failed SSH login attempts:

```
#!/bin/bash
day=`date | cut -c 5-10`
count=`grep "$day" /var/log/auth.log | grep "authentication failure" | wc -l`
echo "$day: $count" >> /home/student/failed_auth.log
```

The first line of the script simply tells the interpreter what kind of script it is. You will see this is common in almost all scripts. It starts with `#!` (the "shuh-bang") and is followed by the path to the program you want to interpret the script. In this case, we want to use `/bin/bash` to interpret the script.

The second line of the script uses the `date` command to print out the date, then pipes the date to the `cut` command to pull out just the month and day. Notice that this entire command is enclosed within backticks (`` ` ``). Backticks cause the output of the command to be returned to the script where, in this case, we store them in a variable named `day`.

The third line is more complex. In this line, we use `grep` to search the file `/var/log/auth.log` for lines that contain the month and day stored in the `day` variable. We then do a further `grep` on those lines to look for lines that contain the words "authentication failure," which is indicative of a failed login attempt. Finally, we pipe those lines to the `wc` command, which counts how many lines match both of those considerations and produces a number. Again, the entire expression is enclosed within backticks, and so that number gets stored in the `count` variable.

In the fourth line, we use `echo` to print the two variables `day` and `count` and append that string to the file `/home/student/failed_auth.log`.

The end result is a script that we can run that will count how many failed login attempts have happened so far today and write it to a file.

What would be the optimal time to run this script each day?

To learn more about scripting, there is a good tutorial here:
http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html

Using Cron to automate scripts

Naturally, it would be helpful to be able to automatically run the script on your server each day, so you wouldn't have to log in and run it yourself. There is a system utlity called Cron that allows users to schedule tasks to be run on the system.

To schedule a new task, edit `/etc/crontab` and add a new one-line entry. The first five space-separated values are the minute, hour, day of month, month, and day of week the command should be run. The sixth value is the user the command should be run as. The remaining value is the command to be run. You can use asterisks for any of the first five values to mean "every."

For instance, if you wanted to ping google.com once every 10 minutes, you could add lines such as

```
#m h dom m dow user command
0  * * * * student ping -c 1 google.com
10 * * * * student ping -c 1 google.com
20 * * * * student ping -c 1 google.com
30 * * * * student ping -c 1 google.com
40 * * * * student ping -c 1 google.com
50 * * * * student ping -c 1 google.com
```

If you wanted to take a snapshot of which files were in your home directory at 5:15PM every Tuesday

```
15 17 * * 2 student ls /home/student/ > /home/student/snapshot
```

Exercise: Writing scripts of your own

Part 1: Write a script that prints out "`Hello World`".
Part 2: Write a script to backup `/home/student/lesson1/` into `/home/student/lesson1_backup/`
Part 3: Use Cron to run your backup script every morning at 8:55AM.
Hint: You may need to modify your script to remove the previous backup before creating the new one.

Network scanning

While we're on the topic of making the computer automate things for you, wouldn't it be nice if, when you're looking at another computer on the network and want to figure out its IP address and what ports are open, you could get that without having to manually look at the network traffic with `wireshark` or `tcpdump`? What you need is a **network scanner**.

In particular, we'll use the `nmap` network scanner in this course.

A network scanner is a tool that will send a series of network packets to hosts on a network and see what responds. For instance, if we want to find all the hosts on the 192.168.10.0/24 network, we could run

- `nmap 192.168.10.0/24`

to do a scan of all 254 possible hosts on the network and, for each host found, scan the top 1000 most-used tcp ports to see if they're open and accessible. `nmap` is a very powerful scanning tool, and there are many options for what types of scans you may want to try. Some of the things you can do are:

- Detect hosts via ARP traffic (even if they aren't responding to direct requests)
- Detect open ports, or ports that appear to be intentionally blocked (firewalled)
- Fingerprinting (Detecting which OS version and software versions are running on a host)
- Rate limit your scans to avoid setting off intrusion detection alerts
- Much much more

As always, use the `man` pages to see all the various options for `nmap`.

Firewalls

Say you're configuring a computer and you *know* there will be no good reason for certain network traffic to reach an application, you might use a firewall to block that network traffic.

For instance, say you're running an SSH server so that you can log in to your computer remotely via SSH. We've done this often in this class, SSH'ing from one VM to the next. What if you wanted to allow incoming SSH connections to your computer from the VM's you create, but not from other users on the local physical network? After all, with a guessable username and password of `student/student`, pretty much anybody on the network can log in as you. Dangerous.

Firewalls can work as either a **blacklist** or a **whitelist**. A *blacklist* is a set of rules for blocking network traffic. Anything not blocked is allowed. A *whitelist* is a set of rules for allowing network traffic. Anything not allowed is blocked.

In the above example, we may want to allow all traffic except for the potentially vulnerable SSH service, and then, only block incoming traffic from the local network so that our VM's continue to work.

Let's use `iptables` to create this firewall rule:

- `sudo iptables -A INPUT -i eth0 -p TCP --dport 22 -j REJECT`

In this example, we create a new rule for incoming traffic (`-A INPUT`) that specified that any packets coming in over the `eth0` interface (`-i eth0`) that follows the TCP protocol (`-p TCP`) and has a destination port of 22 (`--dport 22`) should be rejected (`-j REJECT`). By default, all traffic is allowed.

If we had a default `DROP` policy (e.g. a whitelist), we could make a new rule to allow the VM connections like this:

```
sudo iptables -A INPUT -i veth0 -p TCP --dport 22 -j ACCEPT
```

To list our current rules, we can use:

- `sudo iptables -L`

You can filter traffic on many different aspects of network traffic. Want to deny incoming HTTP (port 80) traffic from the host with *source* address 192.168.10.10?

```
sudo iptables -A INPUT -p TCP --dport 80 -s 192.168.10.10 -j REJECT
```

Purely as an example of the power of `iptables`, what if you want to limit incoming SSH connections to a maximum of 3 at a time, from any source:

```
sudo iptables -A INPUT -p TCP --dport 22 -m connlimit --connlimit-above
3 --connlimit-mask 0 -j REJECT
```

Last, to *flush* all the rules from your table:

- `sudo iptables -F`

A word on rejecting packets: When you decide to block packets from a particular source, you have the choice of using either a `DROP` rule or a `REJECT` rule. A `DROP` rule will discard the packet and do nothing else. A `REJECT` rule will discard the packet and send a response to the sender that the packet was refused.

This has a few notable consequences. If you're trying to be stealthy, you may think that a `DROP` rule would keep people from scanning and getting any information from you, because you don't send responses back. However, when you run a blacklist and have most ports open, if a packet comes in destined for an unregistered port (that is, no application is listening on that port), the network stack will typically automatically send a refusal packet back to the sender. For this reason, if your computer is being scanned and is `DROP`ing packets sent to port 22 (SSH), the scanner may see refusal packets from ports 19, 20, 21, … 23, 24, and so on. "Wait. No refusal packet from port 22? Oooooh. They must have a firewall in the way blocking access." This can be an indicator to smart adversaries. In comparison, if you `REJECT` packets sent to port 22, your computer will be indistinguishable from a machine that isn't running an SSH server.

Exercise: Identify hosts and block connections

Part 1: Navigate to `lesson9/exercise1` and run `start.sh` to launch two virtual machines.
Use `nmap` to find their IP addresses.
What services are they running?

Part 2: Run `lesson9/exercise1/enable_firewall.sh` to enable the firewall on one of the VM's.

Which VM is no longer accepting SSH connections from the host?
Can you still SSH to it? What about from the other VM?

Part 3: Run `lesson9/exercise1/initiate_connections.sh`. Every 10 seconds, the two VM's will SSH into the host machine and use the `wall` command to post a message. Use `iptables` to block incoming SSH connections from Client 1, but not Client 0.
Hint: You can use `lesson9/exercise1/firewall.sh` as an example, although you will need to change some of the parameters.

Bypassing the firewall

Firewalls are great tools, but they are not the ultimate protection they may at first seem. There are many ways around firewalls.

Find holes in the perimeter

Often, companies will use a firewall to block off virtually all access to their network, creating a perimeter defense, but will not use firewalls to block traffic within their network. This can make it difficult for an attacker to breach the network, but once she gains access, the firewall no longer stops her from *pivoting* and accessing other machines on the network. Network scanners can often show you holes in the firewall that you can use to get inside the perimeter.

For instance, in Part 2 of the last exercise, the VM blocked SSH access from your host computer, but it didn't block SSH access from the other VM. By gaining access to one VM, you obtained an entrypoint to the machine the firewall was blocking.

Tunnel through

Frequently certain ports are blocked to protect potentially vulnerable services. For instance, your company may be running a web server (port 80) that displays sensitive data and should only be visible to certain IP addresses corresponding to certain computers on the 192.168.30.0/24 network. Let's say your computer's IP address is 192.168.20.7 and therefore the firewall is blocking your access to the web server. However, the firewall is not blocking access to SSH and you have a user account on the server. You can use SSH port forwarding to tunnel your HTTP traffic over port 22 and past the firewall.

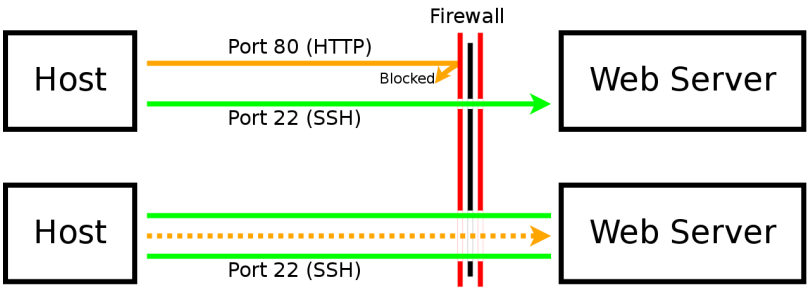


Figure 9.1: Bypassing firewall with SSH tunnel

Initiate the connection from inside

It is very common for incoming traffic to be blocked, but it is often inconvenient to block outgoing traffic. There are so many applications generating packets destined for different ports that both

whitelist and blacklist approaches are often very difficult to apply to outgoing traffic. Taking advantage of the usual lack of an outbound firewall, if you are ever in a position where you have access to the machine already, you can use it to grant inbound access. Generally, this takes the form of a **reverse shell**, where you combine a program like netcat (data transfer over the network) with a program like bash (a shell that gives you access to commands). You can either take advantage of the firewall's blacklist approach and use a non-standard port that isn't blocked, or, if the sysadmin is especially thorough and uses a whitelist, initiate an outgoing connection to set up a data connection. Then you can feed that data connection to your shell to grant remote access, even if normal remote access methods (such as SSH) are blocked.

Exercise: Reverse shell

Navigate to `lesson9/exercise2/`. Take a look at `reverse_shell.sh`.
This script makes heavy use of Linux pipes to connect the output of one program into the input of another one. If you run this script on a machine, it will open up a listening socket on TCP port 4444 and allow *anyone* to connect to it. Then, it will run *as root* any commands that appear on that connection, and pass the output of those commands back over the network connection.

Run `start.sh`. It will launch a VM that is accessible via SSH for 60 seconds. After 60 seconds, a firewall will activate, and any SSH connections (even ones in progress) will be blocked.

Your job is to take advantage of the momentary lapse in security, and start a reverse shell on the VM while you still have SSH access. Then when your SSH access is cut off, use the reverse shell to disable the firewall after it comes up.

Tip: If/when your connection dies and your terminal freezes up, you can press the following keys (in order, not all at once) to kill your `ssh` session and regain control of the terminal:

```
<enter> ~ .
```

Hint: If you run out of time, you can rerun `start.sh` to relaunch the VM and start the countdown over again. You will probably need to restart the exercise more than once as you practice entering the proper commands quickly.

Hint2: Copy and Paste are your friends.

Hint3: Since you are redirecting standard input, you might not be able to type a sudo password. Hmm… How else might you run this command with root privileges?

Lesson 10

Capture the Flag

For our final lesson, there is no new instruction. Simply navigate to `/home/student/lesson10/` and begin the Capture the Flag exercise.